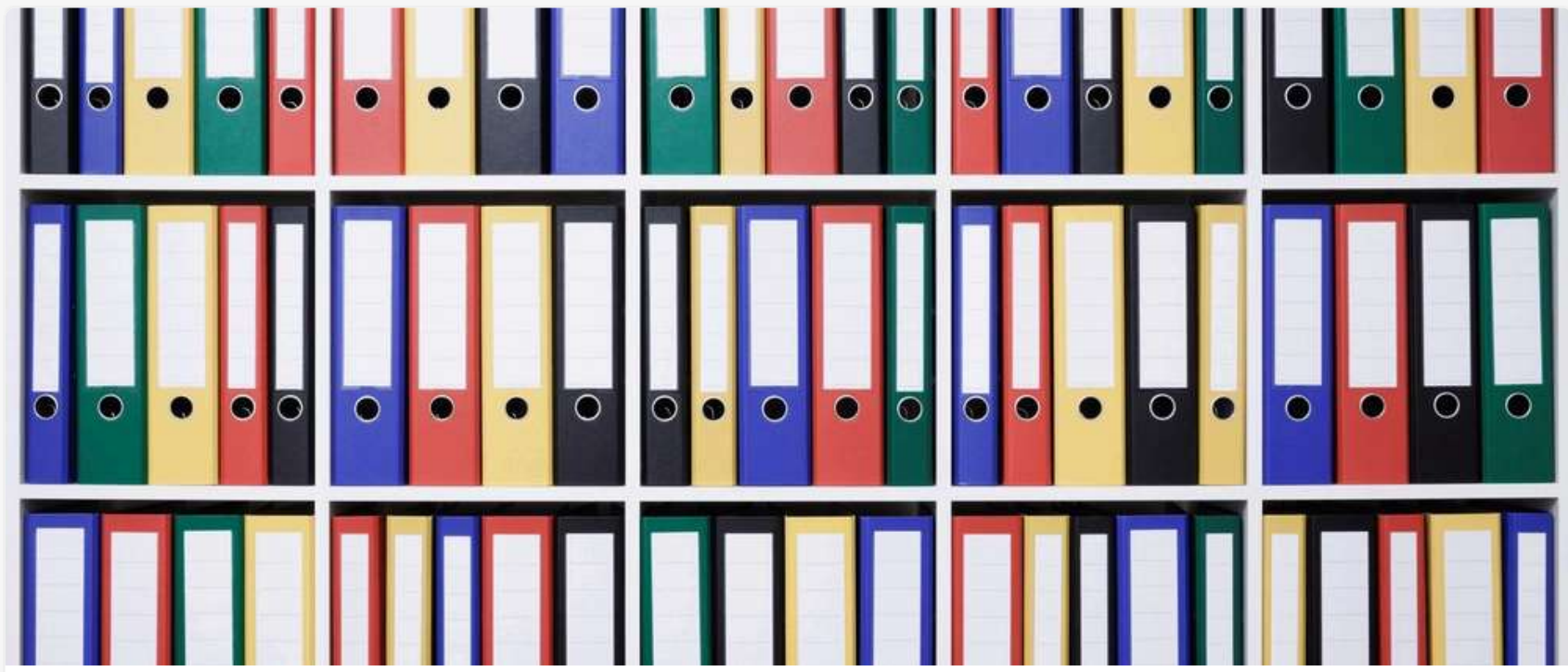

David Whitney
Posted on Apr 17, 2022

Storing Growing Files Using Azure Blob Storage and Append Blobs

#azure #webdev #typescript #architecture

There are several categories of problems that require data to be append only, sequentially stored, and able to expand to arbitrary sizes. You'd need this type of "append only" file for building out a logging platform or building the data storage backend for an Event Sourcing system.

In distributed systems where there can be multiple writers and often when your files are stored in some cloud provider the "traditional" approach to managing these kinds of data structures often don't work well.

You could acquire a lock, download the file, append to it and re-upload – but this will take an increasing amount of time as your files grow, or you could use a database system that implements distributed locking and queuing – which is often more expensive than just manipulating raw files.

Azure blob storage offers **Append Blobs**, which go some way to solving this problem, but we'll need to write some code around the storage to help us read the data once it's written.

What is an Append Blob?

An Append Blob is one of the **blob types** you can create in Azure Blob Storage – Azure's general purpose file storage. Append Blobs, as the name indicates, can only be **appended to** – you append **blocks** to append blobs.

From Azure's Blob Storage documentation:

"An append blob is composed of blocks and is optimized for append operations. When you modify an append blob, blocks are added to the end of the blob only, via the Append Block operation. Updating or deleting of existing blocks is not supported. Unlike a block blob, an append blob does not expose its block IDs."

Each block in an append blob can be a different size, up to a maximum of 4 MiB, and an append blob can include up to 50,000 blocks. The maximum size of an append blob is therefore slightly more than 195 GiB (4 MiB X 50,000 blocks).

There are clearly some constraints there that we must be mindful of, using this technique:

- Our total file size must be less than **195GiB**
- Each **block** can be no bigger than **4MiB**
- There's a hard cap on **50,000 blocks**, so if our block size is less than **4MiB**, the maximum size of our file will be less.

Still, even with small blocks, **50,000 blocks** should give us a lot of space for entire categories of application storage. The Blob Storage SDKs allow us to read our stored files as one contiguous file or read ranges of bytes from any given offset in that file.

Interestingly, we can't read the file block by block – only by byte offset, and this poses and interesting problem – if we store data which has any kind of data format that isn't just plain text (e.g., JSON, XML, literally any data format) and we want to seek through our file, there is no way we can ensure we read valid data from our stored file, even if it was written as a valid block when first saved.

Possible Solutions to the Read Problem

It's no good having data if you can't meaningfully read it – especially when we're using a storage mechanism specifically optimised for **storing large files**. There are a few things we could try to make reading from our **Append Only Blocks** easier.

- We could maintain an index of byte-offset to block numbers
- We could pad our data to make sure block sizes were always consistent
- We could devise a read strategy that understands it can read partial or seemingly malformed data

The first solution – maintaining a distinct index, may seem appealing at first – but it takes a non-trivial amount of effort to maintain that index and make sure that it's keep both in track and up to data with our blob files. This introduces the possibility of a category of errors where those files drift apart, and we may well be in a situation where data appears to get lost, even if it's in the original data file, because our index loses track of it.

The second solution is the "easiest" – as it gives us a fixed block size that we can use to page back through our blocks – but storing our data becomes needlessly more expensive.

Which really leaves us with our final option – making sure the code that reads arbitrary data from our file understands how to interpret malformed data and interpret where the original write-blocks were.

Scenario: A Chat Application

One of the more obvious examples of infinite-append-only logs are chat applications – where messages arrive in a forwards-only sequence, contain metadata, and to add a little bit of spice, must be read **tail-first** to be useful to their consumers.

We'll use this example to work through a solution, but a chat log could be an event log, or a list of business events and metadata, or really, anything at all that happens in a linear fashion over time.

We'll design our fictional chat application like this:

- A Blob will be created for every chat channel.
- We'll accept that a maximum of 50,000 messages can be present in a channel. In a real-world application, we'd create a subsequent blob once we hit this limit.
- We'll accept that a single message can't be more than 4MiB in size (because that'd be silly).
- In fact, we're going to limit **every single chat message to be a maximum of 512KiB** – this means that we **know** that we'll never exceed the maximum block size, and each block will only contain a single chat message.
- Each chat message will be written as its own distinct block, including its metadata.

- Our messages will be stored as JSON, so we can also embed the sender, timestamps, and other metadata in the individual messages.

Our message could look something like this:

```
{
  “senderId”: "foo",
  “messageId”: "some-guid",
  “timestamp”: "2020-01-01T00:00:00.000Z",
  “messageType”: "chat-message",
  “data”: {
    “text”: "hello"
  }
}
```

This is a mundane and predictable data structure for our message data. Each of our blocks will contain data that looks roughly like this.

There is a side effect of us using structured data for our messages like this – which is that if we **read the entire file from the start**, it would not be valid JSON at all. It would be **a text file with some JSON items inside of it**, but it wouldn’t be “a valid JSON array of messages” – there’s no surrounding square bracket array declaration [ ], and there are no separators between entries in the file.

Because we’re not ever going to load our whole file into memory at once, and because our file isn’t actually valid JSON, we’re going to need to do something to indicate our individual message boundaries so we can parse the file later. It’d be really nice if we could just use an open curly bracket { and that’d just be fine, but there’s no guarantee that we won’t embed complicated object structures in our messages at a later point that might break our parsing.

Making our saved messages work like a chat history

Chat applications are interesting as an example of this pattern, because while the data is always append-only, and written linearly, it’s always read in reverse, from the tail of the file first.

We’ll start with the easy problem – adding data to our file. The code and examples here will exist outside of an application as a whole – but we’ll be using **TypeScript** and the **@azure/storage-blob** Blob Storage client throughout – and you can presume this code is running in a modern Node environment, the samples here have been executed in Azure Functions.

Writing to our file, thankfully, is easy.

We’re going to generate a Blob filename from our channel name, suffixing it with **“.json”** (which is a lie, it’s “mostly JSON”, but it’ll do), and we’re going to add **a separator** character to the start of our blob.

Once we have our filename, we’ll prefix a serialized version of our message object with **our separator character**, create an Append Blob Client, and call **appendBlob** with our serialized data.

```
import { BlobServiceClient } from "@azure/storage-blob";

export default async function (channelName: string, message: any) {
    const fileName = channelName + ".json";
    const separator = String.fromCharCode(30);
    const data = separator + JSON.stringify(message);

    const blobServiceClient = BlobServiceClient.fromConnectionString(process.env.AZURE_S

    const containerName = process.env.ARCHIVE_CONTAINER || "archive";
    const containerClient = blobServiceClient.getContainerClient(containerName);
```

```
    await containerClient.createIfNotExists();

    const blockBlobClient = containerClient.getAppendBlobClient(fileName);
    await blockBlobClient.createIfNotExists();

    await blockBlobClient.appendBlock(data, data.length);
};
```

This is exceptionally simple code, and it looks almost like any "Hello World!" Azure Blob Storage example you could thing of. The interesting thing we're doing in here is using a **separator character** to indicate the start of our block.

What is our Separator Character?

It's a nothing! So, the wonderful thing about ASCII, as a standard, is that it has a bunch of control characters that exist from a different era to do things like send control codes to printers in the 1980s, and they've been enshrined in the standard ever since.

What this means is there's a whole raft of character that exist as control codes, that you never see, never use, and are almost fathomably unlikely to occur in your general data structures.

ASCII 30 is my one.

According to ASCII – the 30th character code, which you can see in the above sample being loaded using String.fromCharCode(30), is "RECORD SEPARATOR" (C0 and C1 control codes - Wikipedia).

"Can be used as delimiters to mark fields of data structures. If used for hierarchical levels, US is the lowest level (dividing plain-text data items), while Record Separator, Group Separator, and File Separator are of increasing level to divide groups made up of items of the level beneath it."

That'll do. Let's use it.

By prefixing each of our stored blocks with this invisible separator character, we know that when it comes time to read our file, we can identify where we've appended blocks, and re-convert our "kind of JSON file" into a real array of JSON objects.

Whilst these odd control codes from the 1980s aren't exactly seen every day, this is a legitimate use for them, and we're not doing anything unnatural or strange here with our data.

Reading the Chat History

We're not going to go into detail of the web applications and APIs that'd be required above all of this to present a chat history to the user – but I want to explore how we can read our Append Only Blocks into memory in a way that our application can make sense of it.

The Azure Blob Client allows us to return the metadata for our stored file:

```
public async sizeof(fileName: string): Promise<number> {
    const blockBlobClient = await this.blobClientFor(fileName);
    const metadata = await blockBlobClient.getProperties();
    return metadata.contentLength;
}

private async blobClientFor(fileName: string): Promise<AppendBlobClient> {
    this.ensureStorageExists();
    const blockBlobClient = this._containerClient.getAppendBlobClient(fileName);
    await blockBlobClient.createIfNotExists();
    return blockBlobClient;
}

private async ensureStorageExists() {
    // TODO: Only run this once.
```

```
        await this._containerClient.createIfNotExists();
    }
```

It's been exposed as a **sizeof** function in the sample code above – by calling **getProperties()** on a blobClient, you can get the total content length of a file.

Reading our whole file is easy enough – but we're almost **never going to want to do that**, sort of downloading the file for backups. We can read the whole file like this:

```
    public async get(fileName: string, offset: number, count: number): Promise<Buffer> {
        const blockBlobClient = await this.blobClientFor(fileName);
        return await blockBlobClient.downloadToBuffer(offset, count);
    }
```

If we pass 0 as our offset, and the content length as our count, we'll download our entire file into memory. This is a terrible idea because that file might be 195GiB in size and nobody wants those cloud vendor bills.

Instead of loading the whole file, we're going to use this same function to parse, backwards through our file, to find the last batch of messages to display to our users in the chat app.

Remember

- We **know** our messages are a maximum of 512KiB in size
- We **know** our blocks can store up to 4MiB of data
- We **know** the records in our file are split up by Record Separator characters

What we're going to do is read **chunks of our file, from the very last byte backwards**, in batches of 512KiB, to get our chat history.

Worst case scenario? We might just get one message before having to make another call to read more data – but it's far more likely that by reading 512KiB chunks, we'll get a whole collection of messages, because in text terms 512KiB is quite a lot of data.

This read amount really could be anything you like, but it makes sense to make it the size of a single data record to prevent errors and prevent your app servers from loading a lot of data into memory that they might not need.

```
    /// <summary>
    /// Reads the archive in reverse, returning an array of messages and a seek `positio
    /// </summary>
    public async getTail(channelName: string, offset: number = 0, maxReadChunk: number =
        const blobName = this.blobNameFor(channelName);
        const blobSize = await this._repository.sizeof(blobName);

        let position = blobSize - offset - maxReadChunk;
        let reduceReadBy = 0;
        if (position < 0) {
            reduceReadBy = position;
            position = 0;
        }

        const amountToRead = maxReadChunk + reduceReadBy;
        const buffer = await this._repository.get(blobName, position, amountToRead);

      ...
```

In this getTail function, we're calculating the name of our blob file, and then calculating a couple of values before we fetch a range of bytes from Azure.

The code calculates the start position by taking the total blob size, subtracting the offset provided to the function, and then again subtracting the maximum length of the chunk of the file to read.

After the read position has been calculated, data is loaded into an ArrayBuffer in memory.

```
...

        const buffer = await this._repository.get(blobName, position, amountToRead);

        const firstRecordSeparator = buffer.indexOf(String.fromCharCode(30)) + 1;
        const wholeRecords = buffer.slice(firstRecordSeparator);
        const nextReadPosition = position + firstRecordSeparator;

        const messages = this.bufferToMessageArray(wholeRecords);
        return { messages: messages, position: nextReadPosition, done: position <= 0 };
    }
```

Once we have 512KiB of data in memory, we're going to **scan forwards** to work out where the first **record separator** in that chunk of data is, discarding any data before it in our Buffer – because we know that from that point onwards, because we are strictly parsing **backwards** through the file, we will only have **complete records**.

As the data before that point has been discarded, the updated "nextReadPosition" is returned as part of the response to the consuming client, which can use that value on subsequent requests to get the history block before the one returned. This is similar to how a cursor would work in a RDBMS.

The bufferToMessageArray function splits our data chunk on our record separator, and parses each individual piece of text as if it were JSON:

```
    private bufferToMessageArray(buffer: Buffer) {
        const messages = buffer.toString("utf8");
        return messages.split(String.fromCharCode(30))
            .filter(data => data.length > 0)
            .map(m => JSON.parse(m));
    }
```

Using this approach, it's possible to "page backwards" thought our message history, without having to deal with locking, file downloads, or concurrency in our application – and it's a really great fit for storing archive data, messages and events, where the entire stream is infrequently read due to it's raw size, but users often want to "seek upwards".

Conclusion

This is a fun problem to solve and shows how you could go about building your own archive services using commodity cloud infrastructure in Azure for storing files that could otherwise be "eye wateringly huge" without relying on third party services to do this kind of thing for you.

It's a great fit for chat apps, event stores, or otherwise massive stores of business events because blob storage is very, very, cheap. In production systems, you'd likely want to implement log rotation for when the blobs inevitably reach their 50,000 block limits, but that should be a simple problem to solve.

It'd be nice if Microsoft extended their block storage SDKs to iterate block by block through stored data, as presumably that metadata exists under the hood in the platform.

Top comments (0)

David Whitney

Software consultant. Bestselling Author. Loves rum, alt culture, games & metal. Formerly Head of engineering, chief technical architect, head principal engineer, lead dev, etc.

LOCATION
London, UK

WORK
Independent Software Consultant at Electric Head Software

JOINED
Apr 21, 2020

More from David Whitney

Context Transference
#architecture #code #programming

7 AI Predictions (AI, We Really Need To Talk: Part 2)
#ai #webdev #mcp #machinelearning

Notes on the Synthesis of Form
#designpatterns #architecture #programming