Melange:
Toward Better Persistent Storage for Serverless Applications

Kristen Eberts and Benjamin D. Lee

May 20, 2019

Abstract

Modern Function-as-a-Service (FaaS) offerings from cloud providers lack the ability to share state between functions. Functions can be terminated at any time, so local storage is inherently ephemeral. A common design pattern is for serverless applications to use a cloud storage layer to hold persistent state. However, current storage offerings present distinct tradeoffs in terms of cost, durability, consistency, and performance. We describe the design of a serverless storage layer that could autoscale along multiple dimensions within the tradeoff space. Implementing such a design within modern services has many constraints and challenges; we discuss possible solutions and current roadblocks.

1 Introduction

In serverless applications, computation and storage are neither co-located nor co-provisioned. As such, they scale independently [1]. Serverless functions are inherently stateless; the local storage they use is ephemeral and not shared between functions. Therefore, they must store durable state in a persistent storage layer [2].

Serverless computation is often used for applications with unpredictable or highly variable load. This makes it difficult to use a serverful data store; serverful stores must be pre-provisioned to handle expected load. A common design pattern, therefore, is to use a serverless storage layer such as AWS Simple Storage Service (S3) or DynamoDB to store persistent state. However, S3 and DynamoDB exhibit very different tradeoffs in cost and performance. DynamoDB offers lower average latency than S3, but at a higher throughput cost; S3 offers a lower storage cost than DynamoDB, but costs more for equivalent IOPS [3, 4, 5, 6].

Our project examines the possibility of combining S3 and DynamoDB into a flexible serverless object store. Our proposed design, dubbed Melange, would use DynamoDB as a resizable cache for S3. This would permit applications to achieve an acceptable combination of cost, latency and throughput anywhere within the supported range of the underlying services.

Our paper is structured as follows: Section 2 provides background on the problems with serverless durable storage; Section 3 describes a potential solution to that problem, a system we call Melange; Section 4 discusses the challenges we encountered trying to implement Melange; Section 5 discusses our application of Melange

to a real-world system; Section 6 evaluates that application and discusses its drawbacks; and section 7 presents an overview of related work in this area.

2 Background

Serverless computing has two primary components: Functions-as-a-Service (FaaS) and Backend-as-a-Service (BaaS). FaaS is a less mature market segment than BaaS. Amazon S3 provided a "serverless" infrastructure layer several years before the concept of *serverless computing* was introduced by Amazon Lambda [1]. FaaS offerings are more coherent and compatible than BaaS options, perhaps because BaaS was developed before serverless design patterns were common. For example, Amazon's serverless computation offerings consist of Amazon Lambda, Amazon Lambda@Edge (Lambda, but in a CDN) and AWS IoT Greengrass (Lambda, but in your fridge) [7]. The underlying framework is still Amazon Lambda; the difference between these services is location and latency, not functionality.

In contrast, Amazon's BaaS offerings each have a different tradeoff between cost, durability and performance. There is some flexibility within offerings; for instance, S3 allows users to choose different tradeoffs based on their expected usage patterns [6]. But DynamoDB and S3, or S3 and Amazon Elastic File System, cannot interface with each other as easily as Amazon Lambda can interface with Lambda@Edge and Greengrass. (See Table 1 for an explanation of the differences between CRUD operations in S3 Select and DynamoDB.)

Furthermore, BaaS options have different assumptions about consistency, performance and object size. Object stores are easy to scale and inexpensive for data at rest. However, retrieving data is usually a high-latency, expensive operation. Because of this, attaining high IOPS from an object store like S3 is very costly. Conversely, serverless key-value stores like DynamoDB are capable of very high throughput–but have much higher storage costs for data at rest [1]. Currently, there is no easy way to combine multiple cloud storage offerings into a single, flexible storage layer that mitigates the downsides of the underlying technologies.

What we described above is an optimization problem, one that was solved decades ago by the introduction of memory hierarchies. The "ideal" hardware storage device would be very inexpensive, very fast and very large. However, no technology can be superior on all three dimensions. Every storage option has tradeoffs: SSDs are faster than hard disks but also more costly for the same capacity; SRAM has higher throughput and lower latency than DRAM but is so expensive that it's typically only used for registers. However, by combining multiple storage options you can build a system that is closer to the storage ideal than any of the underlying technologies is [8]. That fact is the core premise of Melange.

3 Melange: The Dream

It seemed logical, based on our background knowledge, to suggest a new BaaS offering that uses the memory hierarchy principle to overcome the limitations of individual storage technologies. Essentially, we wanted to make a version of ElastiCache that is *truly* elastic. In its most ambitious form, our proposed solution would contain a large authoritative stor-

age layer and an faster-but-costlier autoscaling cache layer. Melange's control plane would automatically scale the cache layer up to satisfy performance demands or down to meet cost constraints. If it were implemented by a cloud storage provider, it would use a combination of SSDs (slow, cheap, large) and DRAM (fast, expensive, small) to provide a persistent cache/storage layer for serverless applications.

Unfortunately, the authors of this paper are not a cloud storage provider. So instead of implementing our design on SSDs and DRAM, we tried to implement it on S3 and DynamoDB. The title of the next section, "Challenges," hints at our results.

4 Challenges

Implementing a serverless multi-service CRUD data store requires a significant amount of development to reconcile the incompatible APIs. Each of the three serverless data stores we tested (S3, Aurora Serverless, and DynamoDB) has a completely different interface. Aurora Serverless operates as a normal SQL database, while S3 Select uses a limited subset of SQL [9] and DynamoDB is a NoSQL key-value and object store.

S3 select is one of the most interesting offerings on the market due to the fact that it is not explicitly a serverless database offering, although it can be used as one in certain cases. These cases are limited, as S3 Select's queries are themselves limited to the `SELECT` statement and the `FROM`, `WHERE`, and LIMIT clauses. Subqueries and joins are not supported, nor is data modification. As such, S3 Select is only feasible as a write-once read-many storage system.

By far the most fully featured serverless data store is AWS Aurora Serverless. However, by Amazon's own admission, Aurora Serverless's use case is for variable or infrequent loads. This is due to the fact that while "serverless", Aurora Serverless is a very crude implementation of a serverless computing system. Basically, Amazon turns the server on and off in response to demand. After a query has been completed, the server remains active for a configurable amount of time, currently limited to no less than five minutes. In practice, this system results in a solution that is minimally simpler than the regular serverful Aurora provisioned capacity at the expense of potentially slow scaling (both up and down) [11]. Aurora Serverless also has the "feature" of having a cooldown period for scaling down but no cooldown period for scaling up, and has erratic scaling behavior. For example, a simple `select * from information_schema.tables;` query uses two capacity units instead of one, despite there being no load on the database other than that one query. We therefore eliminated Aurora Serverless as an option because we empirically determined that, when cold, its latency is worse than that of S3 Select (on the order of tens of seconds) and when hot, its latency is comparable to DynamoDB.

DynamoDB is positioned as *the* serverless database offering for AWS. However, its object size limitation of 400 KB results in numerous challenges. Indeed, the DynamoDB best practices documentation specifically recommends the use of S3 to store objects that cannot fit into DynamoDb [10]. However, despite this recommendation, there is no type of official interface between DynamoDB and S3, thereby requiring each developer to implement their own. In an effort to characterize the intersection between S3 Select and DynamoDB, we attempted to implement a shim that made use of DynamoDB for

Action	S3 Select	DynamoDB
Create data	Create a bucket and upload to the bucket in any standard tabular format (*e.g.* CSV)	Create a table and put a JSON-esque document in the table via API
Read data	Submit bucket identifier, filename, SQL query, and input/output serialization methods	Submit a NoSQL JSON query in DynamoDB-specific format
Update data	Not supported in-place; must upload a new version	Submit JSON containing unique key and fields to change in DynamoDB-specific format
Delete data	Automatic deletion as a configurable bucket policy or manual deletion via API	Automatic deletion via a declared integer time-to-live attribute or manual deletion via API

Table 1: While both S3 with S3 Select and DynamoDB offer CRUD operations, their APIs are drastically different [9, 10].

frequently accessed data and S3 Select for large and infrequently used data.

5 Implementation

As an experiment, we attempted to implement a Melange-based system for DNA sequence visualization. Specifically, we modified the DNAvisualization.org serverless architecture [12] to support not only S3 Select but also DynamoDB.

DNAvisualization.org's basic premise is that it takes user-submitted DNA sequences, which can range in size from one letter (or base) to 4.5 million bases, and transforms them into two-dimensional visualizations for interactive exploration. Due to the nature of DNA in which single bases can have a drastic impact (the mutation that causes sickle-cell anemia is caused by a single-base difference), every base must be represented in the visualization. However, as the number of bases increases, it is not possible to display them all at once. Therefore, DNAvisualization.org performs the transformation and stores the visualization as a list of x and y coordinates in S3. When a user zooms in on a region, their request is routed to a Lambda which queries S3 via S3 Select for the coordinates in the new region, downsamples the data to prevent overloading the browser, and returns the data to the client.

This architecture allows for the display of DNA sequences whose lengths span six orders of magnitude. However, the use of S3 as the sole data store presents usability issues. While large sequences are well served by S3, which has enormous bandwidth for uploads, S3 is not ideal for storing the transformations of small sequences, which can comfortably fit in DynamoDB (or po-

tentially the user's browser). Additionally, when zooming in closely into a small region, the system still routes queries to S3, despite the fact the data was already fetched. We therefore implemented a Melange cache for DNAvisualization.org using DynamoDB.

Our implementation of Melange for DNA visualization centers around making the sequence x range query API of the site faster without requiring any changes to the client. In essence, when a user requests a given x range for a sequence ID, Melange first checks to see if the size of the x range puts it into the range of sequence lengths capable of being stored in DynamoDB. We empirically determined this this length to be 12,000 bases of DNA stored as a list of maps. If the DNA could potentially be in DynamoDB, we attempt to fetch it and, if there is any data stored in DynamoDB matching the sequence ID, validate that the data contains the requested x range. If there is, Melange extends time to live (TTL) of the object by one hour. If the is not any data or the data does not match the requested x range, Melange falls back on S3, which stores all of the data. Once the requested x range is located, if the x range can fit into DynamoDB, it is cached with a one hour TTL and returned to the user.

6 Evaluation and Discussion

We find that Melange is successful in reducing query latency from 8 seconds for a 150 base subsequence of a 4.1 million base sequence to 61 ms from DynamoDB. Furthermore, our implementation of Melange is capable of correctly identifying when a requested region is a subset of the region stored in DynamoDB.

Despite successfully implementing a proof-of-concept of Melange, there is presently no compelling reason to use it over existing web technologies for DNAvisualization.org. Specifically, the small size of DynamoDB documents limits its usefulness as a cache for S3. The maximum object size in S3 is 5 TB, whereas the largest object size supported in DynamoDB is 400 KB [13]. Because the objects are limited to storing 12,000 bases of DNA, the latency incurred by invoking a Lambda via API Gateway, querying DynamoDB, and updating the TTL for a stored object is greater than the use of standard web storage technologies such as the NoSQL IndexedDB [14], which supports greater than 10 MB of storage [15]. As such, for this application, the use of DynamoDB increases the cost of operation without increasing performance. Therefore, future implementations of DNAvisualization.org will focus on the use of caching in the browser rather than in DynamoDB. Indeed, the algorithm described in the implementation could be implemented equally well in the browser without incurring the cost of execution in a Lambda. Unfortunately, not all users of serverless computing are connecting via browser with local storage on which to rely. As such, the need for a service which can handle a wide range of data scales is still applicable.

7 Related Work

The Pocket serverless storage service, like Melange, places state on different tiers based on performance and cost criteria. However, Pocket is targeted toward ephemeral computation workloads, and is more fault-tolerant than Melange. It uses distributed state to parallelize computations, whereas Melange's distributed state is organized in a cache hierarchy [13].

Facebook's use of `memcached` to create `memcache`, a distributed key-value store with a look-aside cache, was very influential in our early attempts at building Melange. Like Melange, `memcache` treats the main data store as the authoritative version [16]. Unlike `memcache`, Melange benefits from auto-scaling infrastructure and need not manually partition clusters.

The Anna distributed key-value store is a highly advanced system that uses multiple actors to modify state that is stored in one or more multi-core machines. Anna is not implemented in a serverless style, though it is extremely scalable. Anna is a more advanced version of our original ambition for Melange, which was to create a highly-flexible KVS. Anna's coordination-free model, use of lattice data structures, and consistency assumptions are all more advanced than our design [17].

The most closely related project to Melange is Tiera, a multi-tiered cloud object store. Melange shares several key design characteristics with Tiera, with the notable difference that Tiera uses serverful storage tiers and requires clients to attach provisioned instances of their desired tiers. Like Melange, Tiera is restricted to an object storage model because it is the only model supported by all of its underlying tiers. The Tiera design offers inspiration for future work on Melange. For example, applications can assign tags to objects in Tiera and declare policies for objects with certain tags [18].

Our proposed Melange design, as well as the related works discussed above, could mitigate some of the constraints of existing serverless storage options. But none of them address a core limitation of serverless functions: the fact that function instances cannot maintain persistent local state. That may be a temporary problem, though. Azure Durable Functions is a FaaS extension that permits functions to call other functions directly, and guarantees that local state will be preserved in the event that a process recycles or its container is rebooted. However, Durable Functions must be explicitly orchestrated, and does not automatically cache responses [19]. We hope that other cloud providers will follow suit and enable durable local storage for serverless functions.

8 Conclusion

Persistent storage for serverless apps is a persistent problem for serverless developers. We propose a better solution than the current serverless storage offerings, a system called Melange that would be capable of seamlessly integrating the various data storage options on the market. We tested a prototype using S3 as the long-term and large-object storage level with DynamoDB acting as a cache on an existing serverless web app architecture. We found that DynamoDB's document size restrictions are such that it is presently no better than browser-based storage for the application we tested. We believe that this problem is solvable, but not by combining existing serverless storage products. A superior serverless storage product would have to be developed from the hardware up so that the storage tiers are mutually compatible. We hope such a service will become available in the future.

The source code for our prototype may be found at https://git.io/fj8Ny.

References

[1] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Tech. rep. UCB/EECS-

2019-3. University of California at Berkeley, 2019. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf.

[2] *Programming Model - AWS Lambda*. 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html.

[3] *Amazon S3 Pricing*. 2019. URL: https://aws.amazon.com/s3/pricing/.

[4] *Amazon DynamoDB Pricing for On-Demand Capacity*. 2019. URL: https://aws.amazon.com/dynamodb/pricing/on-demand/.

[5] *Read/Write Capacity Mode - Amazon DynamoDB*. 2019. URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html.

[6] *Object Storage Classes - Amazon S3*. 2019. URL: https://aws.amazon.com/s3/storage-classes/?nc=sn&loc=3.

[7] *Serverless Computing fffdfffdfffd Amazon Web Services*. 2019. URL: https://aws.amazon.com/serverless/.

[8] David Money Harris and Sarah L. Harris. *Digital Design and Computer Architecture*. Second edition. New York, NY, USA: Morgan Kaufmann, 2013. ISBN: 978-0-12-394424-5.

[9] *SQL Reference for Amazon S3 Select and Glacier Select*. 2019. URL: https://docs.aws.amazon.com/AmazonS3/latest/dev/s3-glacier-select-sql-reference.html.

[10] *Best Practices for Storing Large Items and Attributes*. 2019. URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-use-s3-too.html.

[11] *How Aurora Serverless Works*. 2019. URL: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless.how-it-works.html.

[12] Benjamin D. Lee, Michael A. Timony, and Pablo R. Ruiz. "DNAvisualization.org: A Serverless Web Tool for DNA Sequence Visualization". In: *Nucleic Acids Research* (2019). DOI: 10.1093/nar/gkz404.

[13] Ana Klimovic et al. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '18. Carlsbad, CA: USENIX Association, 2018, pp. 427–444. URL: https://www.usenix.org/system/files/osdi18-klimovic.pdf.

[14] *ndexed Database API 3.0*. 2019. URL: https://w3c.github.io/IndexedDB/.

[15] *Browser storage limits and eviction criteria*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria.

[16] Rajesh Nishtala et al. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*. NSDI '13. Lombard, IL: USENIX Association, 2013, pp. 385–398. URL: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf.

[17] Chenggang Wu et al. “Anna: A KVS For Any Scale”. In: IKDE (Feb. 2019). URL: http://db.cs.berkeley.edu/jmh/papers/anna_ieee18.pdf.

[18] Ajaykrishna Raghavan, Abhishek Chandra, and Jon B Weissman. “Tiera: Towards Flexible Multi-Tiered Cloud Storage Instances”. In: *Proceedings of the 15th International Middleware Conference*. Middleware ’14. Bordeaux, France: ACM, 2014. DOI: 10.1145/2663165.2663333. URL: http://dcsg.cs.umn.edu/Papers/tiera_middleware14.pdf.

[19] *What are Durable Functions?* 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.