Frontiers in High Performance Computing

TYPE Original Research
PUBLISHED 04 September 2023
DOI 10.3389/fhpcp.2023.1127883

EDITED BY
Fred Douglis,
Peraton Labs, United States

REVIEWED BY
Scott Klasky,
Oak Ridge National Laboratory (DOE),
United States
Steve Chapin,
Center for Applied Scientific Computing,
United States

*CORRESPONDENCE
Somali Chaterji
schaterji@purdue.edu

RECEIVED 20 December 2022
ACCEPTED 16 August 2023
PUBLISHED 04 September 2023

CITATION
Shankar K, Mahgoub A, Zhou Z, Priyam U and Chaterji S (2023) Asgard: Are NoSQL databases suitable for ephemeral data in serverless workloads?
Front. High Perform. Comput. 1:1127883.
doi: 10.3389/fhpcp.2023.1127883

Asgard: Are NoSQL databases suitable for ephemeral data in serverless workloads?

Karthick Shankar[1], Ashraf Mahgoub[2], Zihan Zhou[3], Utkarsh Priyam[3] and Somali Chaterji[3,4]*

[1]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, United States, [2]Department of Computer Science, Purdue University, West Lafayette, IN, United States, [3]Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, United States, [4]Department of Agricultural and Biological Engineering, Purdue University, West Lafayette, IN, United States

Serverless computing platforms are becoming increasingly popular for data analytics applications due to their low management overhead and granular billing strategies. Such analytics frameworks use a Directed Acyclic Graph (DAG) structure, in which serverless functions, which are fine-grained tasks, are represented as nodes and data-dependencies between the functions are represented as edges. Passing intermediate (ephemeral) data from one function to another has been receiving attention of late, with works proposing various storage systems and methods of optimization for them. The state-of-practice method is to pass the ephemeral data through remote storage, either disk-based (e.g., Amazon S3), which is slow, or memory-based (e.g., ElastiCache Redis), which is expensive. Despite the potential of some prominent NoSQL databases, like Apache Cassandra and ScyllaDB, which utilize both memory and disk, prevailing opinions suggest they are ill-suited for ephemeral data, being tailored more for long-term storage. In our study, titled Asgard, we rigorously examine this assumption. Using Amazon Web Services (AWS) as a testbed with two popular serverless applications, we explore scenarios like fanout and varying workloads, gauging the performance benefits of configuring NoSQL databases in a DAG-aware way. Surprisingly, we found that, per end-to-end latency normalized by $ cost, Apache Cassandra's default setup surpassed Redis by up to 326% and S3 by up to 189%. When optimized with Asgard, Cassandra outdid its own default configuration by up to 47%. This underscores specific instances where NoSQL databases can outshine the current state-of-practice.

KEYWORDS

serverless computing, DAGs, NoSQL databases, AWS Lambda, Video Analytics, Apache Cassandra, Redis

1. Introduction

Serverless computing (or Function-as-a-Service, FaaS) provides a platform for developers to rapidly prototype their ideas without having to consider administrative tasks like resource provisioning, scalability etc. The cloud provider takes care of these factors on behalf of the users using container-based virtualization, while providing the end user with a simpler interface to be able to focus on the application logic. Besides this, today's popular commercial frameworks like AWS Lambda, Azure Functions, or Google Cloud Functions offer attractive features like per-millisecond billing, elasticity, and automatic scalability based on function inputs (Jonas et al., 2019; Amazon, 2021a; Google, 2021; Microsoft, 2021).

Due to these features, serverless frameworks lend themselves to chained analytics pipelines such as machine learning (Carreira et al., 2018; Rausch et al., 2019), Video Analytics (Fouladi et al., 2017; Ao et al., 2018), or IoT processing (Shankar et al., 2020). These pipelines often make use of Directed Acyclic Graphs (DAGs) to lay out the functions of the application in different stages (Daw et al., 2020; Bhasi et al., 2021;

Mahgoub et al., 2021, 2022a,b; Zhang et al., 2021). In such cases, data has to be passed between the various functions in a DAG. Today, FaaS platforms rely on asynchronous communication mechanisms, where data between dependent functions is often passed over the network via remote storage like S3 (Amazon, 2021c) or distributed key-value stores like (Redis, 2021). This intermediate data passed between functions of the DAG is typically short-lived, and hence known as "ephemeral data", and thus, does not require long-term storage. It also does not require reliable storage since this intermediate data can be re-generated by simply re-executing the DAG.

Column-oriented NoSQL databases like Cassandra (2021) or ScyllaDB (2021) are designed to provide a scalable and cost-optimized storage (in $), but they require effective and workload-aware online configuration to sustain the best performance (Klimovic et al., 2018b; Mahgoub et al., 2019). Hence, to date, no study has advocated for the use of NoSQL databases for ephemeral data in serverless workloads. How to handle ephemeral data in serverless DAGs is not just an academic question, but rather it has an important bearing on whether such a platform can support latency-critical applications. For example, Pu et al. (2019) show that running the CloudSort benchmark with 100TB of data on AWS Lambda with S3 remote storage, can be up to 500× slower than running on a cluster of VMs, with most of the additional latency consumed in intermediate data passing (shuffling). Our own experiment with a machine learning application (Figure 3) that has a simple linear workflow shows that intermediate data passing with remote storage takes over 75% of the execution time.

The overarching question that we pose, and partially answer, is:

> Can NoSQL databases be used for ephemeral data in serverless DAGs? Can the already developed and well optimized NoSQL databases along with their automated tuning solutions be used in serverless environments?

We find that for latency-critical applications, we can achieve a better performance per $ (Perf/$) cost[1] with NoSQL databases than with the state-of-practice, either disk-based (as S3) or memory-based (as Elasticache Redis), for serverless ephemeral data. This is possible as NoSQL databases, such as Cassandra, already leverage both memory and disk storage media to deliver optimized performance, and the proportions of these are easily configurable to meet the workload requirements, as shown in Mahgoub et al. (2019); Tran et al. (2008); Gilad et al. (2020); Duan et al. (2009); Curino et al. (2010). Thus, if the databases are tuned to the right configuration, based on the workload characteristics, we can achieve a better operating point in terms of latency and cost.

Optimizing NoSQL databases for static workloads has been explored in prior works like OtterTune (Van Aken et al., 2017), Rafiki (Mahgoub et al., 2017), BestConfig (Zhu et al., 2017), OptimusCloud (Mahgoub et al., 2020), and (Sullivan et al., 2004). Other prior works took a step further to reconfigure databases in response to changing workloads in ScyllaDB (Scylla, 2018) and SOPHIA (Mahgoub et al., 2019). If NoSQL databases can indeed be configured for use here, this would serve as an additional option to the state-of-the-practice remote storage with Amazon S3 (Amazon, 2021c) or state-of-the-art with Pocket (Klimovic et al., 2018b) or local VM storage with SAND (Akkus et al., 2018). Such NoSQL optimization techniques can be employed in a serverless setting for ephemeral data. This has the potential to improve matters if the databases were tuned *in a DAG-aware manner*, such as the degree and type of fanout (e.g., scatter or broadcast) and the volume of intermediate data. Furthermore, a fanout in the DAG with a high degree of parallelism can take advantage of the high read throughput provided by replicated NoSQL databases.

To optimize the data store for serverless functions, we introduce Asgard[2] that deploys the popular NoSQL database, Cassandra, for ephemeral data storage and compares with the current states-of-practice (S3 and ElastiCache Redis). We also compare Cassandra with static and dynamic reconfiguration [following the guidance of SOPHIA (Mahgoub et al., 2019)] on the overall Perf/$[3]. We call the former Asgard Lite and the latter simply Asgard because Asgard can work for dynamic workloads, as is more commonplace in the real world, but comes with more engineering overhead. We find that Cassandra with its default configuration has up to 3.06X better Perf/$ than S3; while optimized Cassandra, such as using Asgard on top, has up to 1.47X better Perf/$ than Cassandra with the default configuration. Thus, based on our evaluation, Cassandra, when properly tuned to the workload characteristics, is able to deliver higher Perf/$ than S3 or Redis, while it delivers E2E latency within 10% of Redis at most. We also discuss the effects of VM placement and dynamic workloads on performance per $ cost. Cassandra with Asgard on a dynamic workload achieves a better performance per $ cost, even with minor changes in the workload characteristics over time.

2. Preliminaries

In this section, we layout the necessary background information for the rest of the paper. We start by summarizing the main features of NoSQL databases and what features they ought to provide for our use case. Next, we summarize the concept of serverless computing and how its performance (i.e., end-to-end latency) relies heavily on the response time of the ephemeral data storage coupled with it. Finally, we state the key features that NoSQL databases provide and from which serverless computing can benefit such as fast scalability and geo-distributed replication.

2.1. NoSQL databases and Cassandra

NoSQL databases is a form of databases that provide considerable durability and performance gains that significantly

1 Here, performance is defined as the inverse of end-to-end execution time (i.e., E2E latency) while cost is the raw $ value of the services on AWS as of November 2022.

2 Asgard is the abode of Gods with special abilities including enhanced senses, such as, detecting that a change is impending.

3 We simplified the full functionality of SOPHIA and adapted it to the serverless context. This included understanding the nature of ephemeral data between serverless DAGs and optimizing based on that, e.g., there are bursty writes and reads and little data reuse. Our contribution is *not* a configuration engine for Cassandra, but rather to answer the "overarching question" posed above.

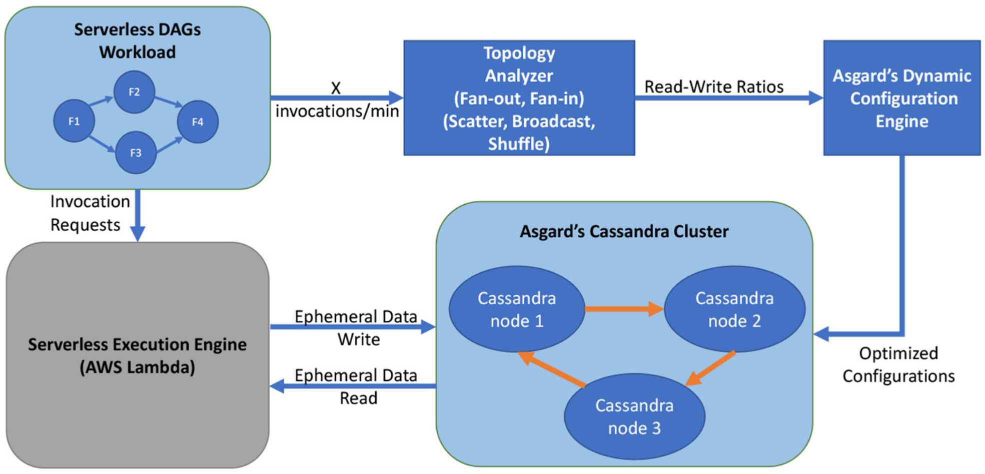

FIGURE 1
Workflow of Asgard. First, a topology analyzer inspects the DAG definitions and extracts the read-write ratios for every stage in the DAG. Second, Asgard's dynamic configuration engine uses the read-write ratio for every DAG stage (weighted by the invocation frequency of every DAG) and identifies the optimized configurations for the ephemeral data storage cluster. Finally, the DAG executes in the execution engine (AWS Lambda) and all ephemeral data read and write operations are performed by Asgard's optimized Cassandra cluster.

exceeds the limits of traditional relational databases. This is achieved through relaxation of one or more of the ACID—atomicity, consistency, isolation and durability—properties. The main advantages of NoSQL databases are scalability and durability, which is delivered through replication. Unsurprisingly, replication further improves data read throughput due to two reasons (1) the ability to place different replicas into different geographic locations, moving data physically closer to distributed readers. (2) Handling network fluctuations that can significantly reduce the throughput (or increase the latency) of one replica, but unlikely to impact all replicas at the same time. Consequently, NoSQL datastores can be a great fit for storing serverless ephemeral data due to their ability to horizontally scale much faster than the current state-of-practice, better handling bursty workloads.

Cassandra, an Apache Foundation[4] project, is one of the leading NoSQL and distributed DBMS driving many of today's modern business applications, including such popular users as Twitter, Netflix, and Cisco WebEx. Cassandra is highly durable and scalable. It also delivers very fast scaling due to its concept of *virtual nodes* (*vnodes*), a data distribution scheme. With *vnodes*, the amount of data that needs to be re-organized across the cluster, in case of adding or removing nodes, is minimized. This is achieved by dividing the data key range into many smaller ranges (*a.k.a.*, tokens) and each node gets an equal, and randomly selected, share of the tokens. When the number of tokens $\gg$ the number of nodes in the cluster, it can be shown that key redistribution time is reduced significantly compared to classic hashing (Röger and Mayer, 2019). Accordingly, Cassandra can be scaled in and out much faster than distributed databases that do not support this feature, including S3 and Redis.

Cassandra overview: key features This section describes the key features of Cassandra that become prime focus areas for performance optimization.

4 http://cassandra.apache.org

2.1.1. Write workflow

Write (or update) requests are handled efficiently in Cassandra using some key in-memory data structures and efficiently arranged secondary storage data structures. When a write request arrives, it is appended to Cassandra's CommitLog, a disk-based file where uncommitted queries are saved for recovery/replay. The commitlog is replayed in case of a fault to revert the node's state back to its state before the fault. After the query is saved into the commitlog, the result of the query is processed into to an in-memory data structure called the Memtable. Then an acknowledgment is sent back to the client. A Memtable functions as a write-back cache of data rows that can be looked up by key, i.e., unlike a write-through cache, writes are batched up in the Memtable until it is full, when it is flushed (the trigger and manner of flushing are controlled by a set of configuration parameters, such as the ones that can be optimized to tune the performance of Cassandra). Each flush operation transfers these contents to the secondary storage representation, called SSTables. SSTables are immutable and every flush task produces a new SSTable. The frequency of performing this flush task needs to be tuned to minimize the number of created SStables.

Another feature of Cassandra is its Scheme Flexibility, which allows users to alter their tables schemes without the need for defining new tables and migrating data. This feature implies that the data for a given key value may be spread over multiple SSTables. Consequently, if a read request for a row arrives, all SSTables (in addition to the Memtable) have to be queried for portions of that row, and then the partial results combined. This prolongs execution time, especially because SSTables are resident in secondary storage. Since SSTables are immutable by design, instead of overwriting existing rows with inserts or updates, Cassandra writes new timestamped versions of the inserted or updated data in new SSTables. Over time, Cassandra may write many versions of a row in different SSTables and each version may have a unique set of columns stored with a different timestamp, which are periodically merged, discarding old data in a process called *compaction* to keep the read operation efficient. A number of optimization techniques are introduced to reduce the possible number of SSTables searches for a read request. Such optimization techniques include: key index caching, Bloom filtering, and compaction.

2.1.2. Compaction

Cassandra provides two compaction strategies, which can be configured on the table level (Cassandra, 2022). The first (and the default) compaction strategy is called "Size-Tiered Compaction", which triggers a new compaction process whenever a number of similar sized SSTables have been created, and "Leveled Compaction", which divides the SSTables into hierarchical levels.

Size-tiered compaction: This compaction strategy activates whenever a set number of SSTables exist on the disk. Cassandra uses a default number of four similarly sized SSTables as the compaction trigger.

While this strategy works well to compact a write-intensive workload, it makes reads slower because the merge-by-size process does not group data by rows.

Leveled compaction: The second compaction strategy divides the SSTables into hierarchical levels, say L0, L1, and so on, where L0 is the one where flushes go first. Each level contains a number of equal-sized SSTables that are guaranteed to be non-overlapping, and each level contains 10X the number of keys at the previous level. Compaction is triggered each time a MEMTable flush occurs, which requires more processing and disk I/O operations to guarantee that the SSTables in each level are non-overlapping. Moreover, flushing and compaction at any one level may cause the maximum number of SSTables allowed at that level (say Li) to be reached, leading to a spillover to Level L(i+1).

Qualitatively it is known DataStax (2022) that size-tiered compaction is a better fit for write-heavy workloads, where searching many SSTables for a read request is not a frequent operation. In contrast, leveled compaction serves as a better fit for read-heavy workloads, where the overhead of guaranteeing the non-overlapping property pays off for frequent read requests. However, determining the appropriate strategy, *together with* the parameters such as size of each SSTable, and conditioning this on the exact workload characteristics (not pure reads or pure writes, but mixes, and their distribution such as Gaussian or Zipfian) requires adaptive configuration tuning such as the one used in our past work (Mahgoub et al., 2019).

2.2. Adaptive configuration tuning

Cassandra is highly tunable, with more than 25 configuration parameters that can alter its performance from write-optimized to read-optimized, or to achieve a balanced performance between the two. For example, in our prior work (Mahgoub et al., 2017), we identified the following key parameters for Cassandra:

1. *Compaction method (CM)*: Categorical value between Size-Tiered (preferred for write-heavy) or Leveled (preferred for read-heavy).
2. *Concurrent writes (CW)*: This parameter gives the number of independent threads that will perform writes concurrently. The recommended value is 8 $\times$ number of CPU cores.
3. *file_cache_size_in_mb (FCZ)*: This parameter controls how much memory is allocated for the buffer in memory that will hold the data read in from SSTables on disk. The recommended value for this parameter is the minimum between 1/4 of the heap size and 512 MB.
4. *Memory table cleanup threshold (MT)*: This parameter is used to calculate the threshold upon reaching which the MEMTable will be flushed to form an SSTable on secondary storage. Thus, this controls the flushing frequency, and consequently the SSTables creation frequency. The recommended setting for this is based on a complex criteria of memory capacity and IO bandwidth and even then, it depends on the intensity of writes in the application. It is recommended to set this value low enough to keep disks saturated during write-heavy workloads (0.11 for HDD, 0.33 for SSD).
5. *Concurrent compactors (CC)*: This determines the number of concurrent compaction processes allowed to run simultaneously on a server. The recommended setting is for the smaller of number of disks or number of cores, with a minimum of 2 and a maximum of 8 per CPU core. Simultaneous compactions help preserve read performance in a mixed read-write workload by limiting the number of small SSTables that accumulate during a single long-running compaction.

Accordingly, we can adjust Cassandra's performance, such as using the impactful parameters shown above, using our prior knowledge of the serverless DAG topology to deliver the maximum throughput and minimum response time. In the serverless context, if the DAG has a broadcast fanout stage of size *k*, then we know that the output of the previous stage will be written once (a single writer), and then read by *k* parallel functions (multiple readers). Hence Asgard (Figure 1) tunes Cassandra for high read throughput and low read latency. On the other hand, when the DAG has a fan-in stage, we have multiple writing functions and a single reader. Accordingly, we tune Cassandra for maximum write throughput and minimum write latency. Finally, since Cassandra has already been studied in prior works (Mahgoub et al., 2019, 2020) and its performance is shown to be superior, given workload variability, Cassandra, especially with this configuration tuning knobs, will be a great fit for our use case of serverless DAGs. Note that the DAG topology is defined by the user before its actual execution. This allows us to anticipate the read/write pattern for the DAG early enough to properly configure Cassandra at runtime. This, in addition, to our ability to tune the topology of the DAGs themselves, will result in a large optimization space of tuning DAG topologies on the one hand, such as using a combination of bundling and fusion of DAG stages, as in Mahgoub et al. (2022b), and tuning the data store on the other hand, as proposed here.

2.3. Serverless computing

In serverless computing, the cloud provider performs all the administrative tasks pertaining to the physical resources on behalf of the user. These tasks include scaling, maintenance, and physical

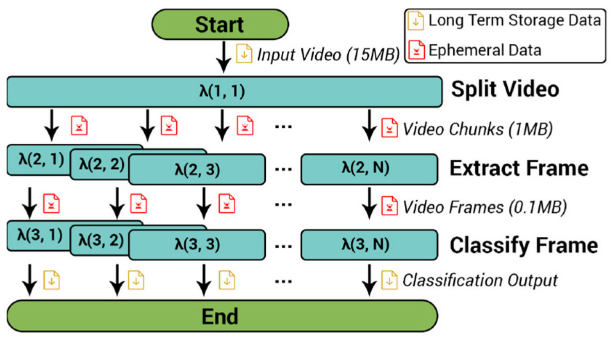

FIGURE 2
DAG overview for Video Analytics application.

scheduling (i.e., deciding which physical machine will host a particular serverless function). This model has the clear advantage of its ease-of-use, as customers do not need to worry about allocating too many or too few resources for their applications. Additionally, the cloud provider performs the physical resource allocation in a way that is hidden from the user. Accordingly, serverless computing has a pay-as-you-go, fine-grained pricing model where customers are only charged for the exact resource usage of their applications, typically in terms of number of cores, memory capacity, and fine-granular execution time (e.g., 100 ms increments).

From the cloud provider perspective, serverless computing is very beneficial since it delegates the resource allocation and mapping to physical machines tasks to the cloud provider. Moreover, serverless functions are usually transient with a small memory footprint (e.g., AWS Lambda's memory size starts from 128 MB). Accordingly, the cloud provider can better schedule the serverless functions dynamically across different physical machines to maximize utilization. Further, since the functions are stateless, i.e., cannot store any data locally for subsequent invocations to use, migrating functions across physical machines at runtime is seamless. However, scheduling functions that have data dependencies among them—such as for functions in a DAG—on different physical machines has negative implications on the data passing time. This is because the functions have to communicate through a remote storage, causing data to be transferred over the network twice, which can significantly increase the DAG's end-to-end latency (Mahgoub et al., 2021). Accordingly, it is essential to minimize the ephemeral data transfer latency to minimize the DAG's end-to-end latency without increasing the execution cost (in $). Notice that the data upload and download operations are performed by the sending and receiving functions. Hence, an increase in the data passing time will also increase the functions' runtime and $ cost.

2.4. Desired features of datastores for serverless systems

Passing ephemeral data between two stages of a serverless DAG is challenging since direct point to point communication between two individual running functions is difficult since communication parameters like IP addresses are not exposed to the users (Mahgoub et al., 2021). As such, the most common method is to use an external remote storage as noted above. One of the common choices, Amazon S3, offers ease of use and fault tolerance at the cost of high latency transfers. As noted in Klimovic et al. (2018b), fault tolerance is not as highly desired in serverless contexts for ephemeral data as compared to traditional storage systems since serverless functions are designed to be idempotent. Lower latency and higher bandwidths are preferred due to the high volume of invocations.

2.5. NoSQL key features for serverless computing

NoSQL databases provide key features that are very useful for serverless computing in general, and for serverless DAGs in particular. First, Serverless workloads are bursty in nature (Shahrad et al., 2020). Accordingly, the ephemeral data storage should be able to quickly scale up and down to adapt to these bursts. As discussed in Section 2.1, Cassandra's adaption of *vnodes* and consistent hashing serves this purpose adequately. Second, the cloud service provider can quickly vary the physical assignment of a serverless function across physical machines and possibly even data centers that reside in different geographic locations. Accordingly, the cloud provider can leverage the ability of Cassandra's geo-distributed multi-datacenter replication capabilities to deliver minimized response times (Zafar et al., 2016). Furthermore, the cloud provider can launch a new replica in a the new datacenter *before* scheduling the functions to that datacenter—in a manner that is analogous to cache pre-fetching—to avoid increased latency during database warm-up periods. Third, since the topology of serverless DAGs is known prior to DAG execution, predicting the workload read/write pattern for ephemeral storage is much easier and can be directly guided by the topology (e.g., fan-out, fan-in, scatter, broadcast, shuffle, etc.). This allows the cloud service provider to efficiently tune the ephemeral storage (Cassandra in our use case) to better handle the dynamic changes in the workload and deliver optimized performance/$.

2.6. Design of Asgard

Asgard is designed on top of SOPHIA (Mahgoub et al., 2019), with changes made to adapt it to serverless workloads. The main contributions of this work is not the system itself but rather to answer the overarching question of if NoSQL databases can be used for serverless workloads. The different components of Asgard as shown in Figure 1 are as follows:

1. **Serverless DAG workloads:** The serverless DAG definition is provided by the user, typically in the form of a JSON (Amazon, 2023a), listing all the different dependencies of the different stages.
2. **Topology analyzer:** The DAG definition is inspected to extract the read-write ratios of the ephemeral data in all the different stages depending on if it is a fan-out or fan-in.

3. **Dynamic configuration engine:** The read-write ratios are used in an optimizer to identify the optimal configurations for the NoSQL database cluster. This configuration is then opportunistically applied to the cluster, similar to the design in SOPHIA (Mahgoub et al., 2019).

3. Experimental setup

We implement our serverless applications on AWS-Lambda (Amazon, 2021b) using different storage systems, while we deploy Cassandra and Asgard in Amazon EC2. We compare the following systems in our evaluation:
Amazon S3 (Amazon, 2021c): Amazon's Simple Storage Service—state-of-practice for storing ephemeral data in serverless.

ElastiCache Redis (Redis, 2021): Amazon's offering of Redis through its ElastiCache service. Redis is also a state-of-practice for in-memory storage, although it has higher costs. The instance used for this is *r5.large*, which belongs to the AWS memory optimized family recommended for Redis Amazon (2022).

Cassandra (default): Vanilla installation of a single node of Cassandra on an *i3.large* EC2 instance. The instance comes with an NVMe SSD storage, which is used to store the data. i3.large is used for this since it is optimized for low latency and very high random I/O performance which makes it a common starting point for Cassandra. This can further be tuned to use an optimized instance type (as referenced in Section 4.3).

Cassandra (Asgard Lite): This is an installation of a single node of Cassandra on an *i3.large* EC2 instance with a reduced functionality version of Asgard (*Lite*), containing the (open sourced) static configuration tuner Rafiki (Mahgoub et al., 2017). The database is tuned with Asgard Lite before running a given application. The *Lite* version is used since we evaluate only on a single application at a given time.

Cassandra (Asgard): This version of Cassandra (also on *i3.large*) has the full Asgard system deployed, which uses a Cost-Benefit Analyzer (CBA) adapted from SOPHIA (Mahgoub et al., 2019), to dynamically switch configurations to maximize Perf/$. Here, the full version is used for a simple pipeline of intermixed Video Analytics and LightGBM applications to demonstrate a dynamic workload. The same Cassandra instance is used for both workloads even though they share no data. This setup has the advantage of amortizing costs across applications by using a multi-tenant model. The *benefit* in CBA is the increase in throughput due to the new optimal configuration over the old configuration:

$$B = \sum_{k \in [0, T_L]} H_{sys}(\mathbf{W}(k), \mathbf{C}_{sys}^{T}(k))$$

where H_{sys} is the throughput of Cassandra given a workload $\mathbf{W}(k)$ and time-varying configuration $\mathbf{C}_{sys}^{T}(k)$. T_L is the look-ahead time for the incoming workload. The *cost* is the transient throughput dip due to reconfiguration:

$$C = \sum_{k \in [1, M]} H_{sys}(\mathbf{W}(t_k), \mathbf{C}_k) \cdot T_r$$

where $\mathbf{C}_k$ is a configuration plan and T_r is the duration Cassandra is offline during reconfiguration.

S3 is configured to be in the same region as the AWS λ functions for the application while Redis and all variants of Cassandra are configured to be in the same VPC as the λ

functions for efficient communication. Note that the availability zone of S3 cannot be configured and we can only specify what region it is in. The data however is replicated across at least 3 availability zones (Amazon, 2021d).

3.1. Applications

To explore the usage of NoSQL databases for ephemeral data in serverless analytics frameworks, we adopt a Video Analytics pipeline from Klimovic et al. (2018b) and a Machine Learning pipeline (LightGBM) from Carreira et al. (2018). These two applications are suitable for serverless workloads as their components can be split into small λ functions that perform tasks in parallel. They also showcase different types of fanout with respect to how data is passed from one stage of the DAG to another—scatter and broadcast. The former means that a larger piece of data is split equally and each split is sent to a child function while the latter means that the same data is sent to all the children functions. The types can impact the database due to their different read-write characteristics. In a scatter fanout, there is a write for each parallel function, followed by a read for each parallel function. In a broadcast fanout, there is only one write, followed by a read for each parallel function. To run the evaluation applications, the AWS Lambda framework is used with Python as the runtime environment.

3.1.1. Video Analytics

The Video Analytics application (whose DAG structure is shown in Figure 2) performs object classification for different frames of a video. In the first stage of the DAG, a λ function splits the input video into n smaller chunks (of size 10s in our evaluation for higher parallelization) and stores them in an ephemeral storage service. It then triggers n λs in the second stage to process these chunks. In this stage, one representative frame is extracted from each chunk and is also stored in ephemeral storage, before triggering a λ for the final stage. In the final stage, each frame is analyzed using a pre-trained deep learning model (MXNet, 2021) and the classification outputs a probability distribution over 1,000 classes. These results can then be stored in a durable, long-term storage. This application is also an example of a scatter fanout as the bigger video chunk is broken down into smaller chunks for individual processing. The dataset for Video Analytics consists of racing videos from YouTube, with classification targets including cars, race-cars, people, and race tracks.

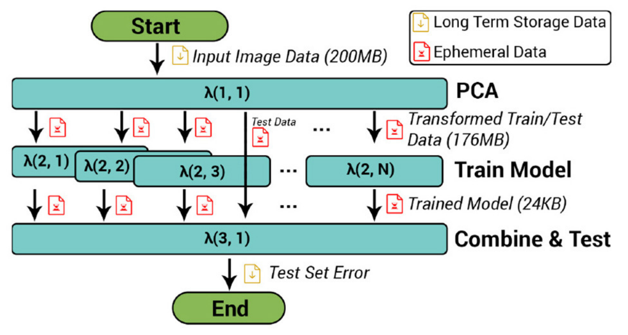

FIGURE 3
DAG overview for LightGBM application.

3.1.2. LightGBM

LightGBM, whose DAG structure is shown in Figure 3, trains an ensemble predictor (a random forest) by training many decision trees. In the first stage, a function performs Principal Component Analysis (PCA) on the input training data and broadcasts this to functions in the second stage, which trains decision trees by using 90% of the data for training, and a random 10%, for validation. The number of functions in this stage is a user parameter (set to six in our evaluation). In the final stage, the model files for these trees are combined to form a random forest to test on testing data. This application is an example of a broadcast fanout since the PCA stage broadcasts the same data to every function in the second stage to train the random forest. The NIST (Grother, 1995) and MNIST (Deng, 2012) datasets are used for this application. Both datasets contain images of handwritten digits. These images are then pre-processed into matrices with grayscale values before being split into a training/testing sets. The first stage of the DAG (PCA) is run with these two sets to get a PCA-transformed dataset. The results of classification are the predicted numbers for the testing set.

4. End-to-end evaluation

In this section, we show the superiority of Asgard over state-of-practice serverless ephemeral storage options (S3 and Redis). In all experiments, we denote Perf/$ to refer to throughput normalized by dollar cost. We keep the configurations (number of cores and memory sizes) for the serverless functions the same across all storage options. This is done to factor out any possible differences in latency or throughput that is not due to storage.

4.1. Video Analytics

In this application, we see from Figure 4 that both variants of Cassandra outperform the state-of-practice (S3) in both Perf/$ and E2E latency. Cassandra with its default configuration outperforms S3 by 189% and Redis by 326% in the Perf/$ metric. With a static configuration tuner configuring Cassandra for this Video Analytics application (taking into account the scatter fanout and 50-50 read-write split), we see that the Perf/$ is 47% better than the default configuration. In terms of the E2E latency, Cassandra with Asgard is 150% faster than S3 and 106% faster than Redis. Also note that Redis has a 21% faster E2E latency than S3, but the overall cost leads its Perf/$ to be 118% lesser than S3. Redis is slower than Cassandra even though it has an in-memory store since the ephemeral data in this application are small chunks of videos and images with each item being accessed only once. Each shard in a replication group has only one read/write node and up to five read-only nodes in AWS ElastiCache Redis (Amazon, 2023c). Thus, for an application with a scatter fanout and a fairly balanced read-write ratio, NoSQL databases are a viable option for ephemeral data.

4.2. LightGBM

We see from Figure 5 that both variants of Cassandra outperform S3 (by around 32%) in the Perf/$ metric. Redis has the highest Perf/$ for this application, being 16% better than Cassandra with Asgard Lite. This is because the application uses a broadcast fanout with many parallel reads of the same object. Redis can handle this better due to its *r*5.*large* instance being a memory-optimized instance to support these operations. In contrast, Cassandra is deployed on an *i*3.*large* instance with an SSD, which is slower than reading directly from memory. S3 has the lowest Perf/$ in this application because the data access latency is the highest.

In terms of E2E latency, we see that Redis has the lowest latency while S3 has a similar latency as that of Cassandra with Asgard Lite. This is because of the structure of the ephemeral data. In LightGBM, the ephemeral data volumes are large and these are read in parallel, which is better handled by S3. Cassandra can also be deployed as a cluster to exploit parallel reading, but the cost tradeoffs for the extra nodes is not worth the small increase in read throughput. Another extension is to dynamically tune Redis using the same principle that we have used here in Asgard. Thus, NoSQL databases are suitable for applications with a broadcast fanout as well.

4.3. Instance placement of Cassandra

In this micro-experiment, we fix the database to be a default installation of Cassandra while we vary the EC2 instance it is placed on. We consider three popular options for the installation of Cassandra, namely *i*3.*large* (instance with a dedicated NVMe SSD disk), *m*5.*large* (a general-purpose instance), and *c*5.*large* (a compute-optimized instance). Figure 6 shows the two metrics (Perf/$ and E2E latency) for Video Analytics and Figure 7 shows the metrics for LightGBM. We can see that for Video Analytics, the compute-optimized *c*5.*large* instance has the best Perf/$ (75% better than *m*5.*large* and 29% better than i3.large). This can be borne out by the scatter fanout of the Video Analytics application. Each item of ephemeral data is written once and read once. Thus, it turns into a relatively more compute-intensive task with multiple queries. *m*5.*large* has the lowest Perf/$ since each read and write is done over the network on EBS storage without the benefits of the compute-optimized cores. For LightGBM, however, we see that *m5.large* has the highest Perf/$ (8.9% better than i3.large and 8.1% better than c5.large). This can be explained due to the broadcast fanout of LightGBM. The same transformed PCA files are read in parallel which benefits the network based EBS storage. Furthermore, *m5.large* instances have a higher memory to compute ratio compared to *c5.large*,

which means that frequently accessed files can be cached more in *m5.large* as opposed to *c5.large*. Thus, this experiment shows the need for a system on top of a database tuner to also tune the EC2 instance that the NoSQL database is installed on. Doing this can lead to the best overall Perf/$.

4.4. Dynamic workloads

In this experiment, we show the benefits of using a dynamically tuned Cassandra node, as opposed to a default static node. We use a combination of the two applications, Video Analytics and LightGBM, in bursts of five serial invocations each. Video Analytics, due to its scatter fanout, has a configuration optimized for 50% write and 50% reads. LightGBM, on the other hand, is a read-heavy application (with large read sizes) due to its broadcast fanout and thus is optimized for a 100% reads. In a real world setting, there could be a variety of invocation patterns with the two applications being interleaved differently. Asgard, using its CBA reconfigures the database based on the incoming workload (e.g., scatter vs. broadcast) and hence optimizes the Perf/$ over all phases of the workload. It may decide to leave the same configuration to avoid the reconfiguration cost if it predicts the new workload to be short-lived. From Figure 8, we can see that there is only a slight gain (2.6% increase in Perf/$ and 1.3% decrease in E2E latency) by dynamically tuning the database. However, this can save the end user thousands of dollars if implemented at a large scale. Also, varying request patterns, within a single application, can be handled well with the dynamic optimization. Furthermore, the figure shows much smaller error bars for Cassandra with Asgard, which shows the stability of the performance under our auto-tuning solution. Here we have not shown the comparison with the non-Cassandra baselines as our prior experiments (Figure 5) already demonstrated the superiority of vanilla Cassandra. The comparison with a static configuration (Asgard-Lite) is also omitted for the same reason. According to Mahgoub et al. (2019), a dynamic configured database achieves 17.6% higher throughput than the static optimized configuration.

4.5. Discussion: what conditions make NoSQL suitable for serverless?

As seen in the evaluation, there are many dimensions of consideration. In this section, we briefly summarize each dimension and its suitability to NoSQL databases.

1. DAG fanout: The current state of practice, S3 (or any other blob storage), performs poorly with scatter fanout due to its fixed strong read-over-write consistency (Amazon, 2023b) which doubles write time. NoSQL stores can be tuned to different levels of consistency based on the read-write characteristics of the application. This makes it better for scatter fanout. For broadcast fanout, or a heavy-read application, performance is more comparable between all the baselines.

2. Data size: Ephemeral data sizes can vary between a few KB to hundreds of MB between stages of a DAG. For applications with larger sizes, S3 has a lower cost of adoption due to the included replication and parallel reads but a NoSQL store with replication can have better performance.

3. Instance placement: While NoSQL databases already have marginal performance improvement over the current state of practice, tuning the instance that the database is in can have further performance benefits as shown in Section 4.3. On average, SSD based instances with high network bandwidth are more expensive but offer lower latency. This consideration is left to an orthogonal work or to the application developer based on the characteristics of the application.

4. Workload Rate: NoSQL databases perform better here with bursty serverless workloads since they can be scaled in or out faster.

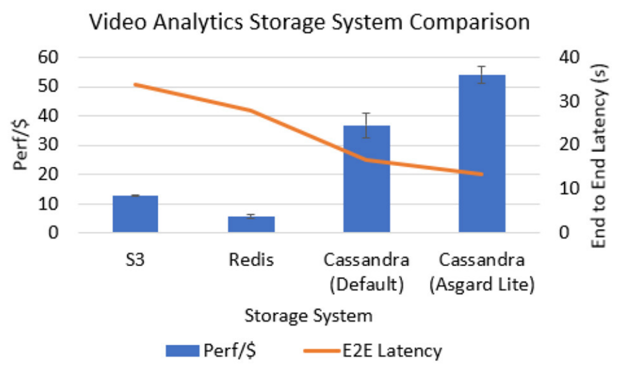

FIGURE 4
Comparison of NoSQL databases vis-à-vis S3 for Video Analytics. Asgard Lite has higher Perf/$ & E2E latency than all baselines due to the scatter fanout attribute of Video Analytics.

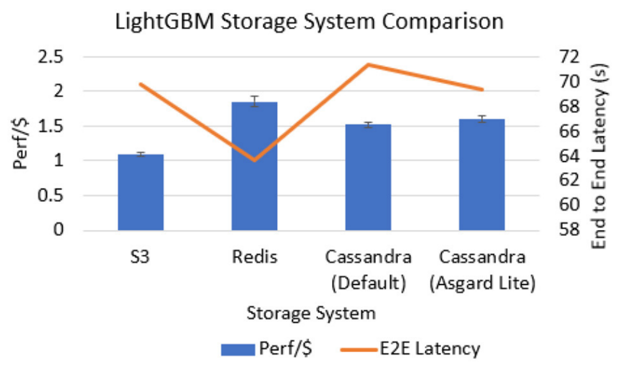

FIGURE 5
Comparison of NoSQL databases vis-à-vis S3 for the LightGBM application. Cassandra outperforms S3 in terms of Perf/$ while Redis outperforms all baselines due to the broadcast fanout attribute of LightGBM.

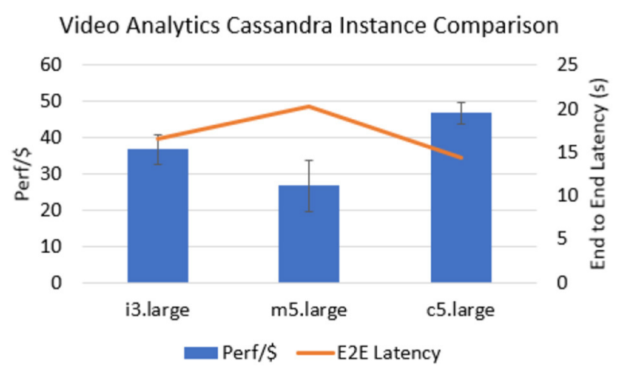

FIGURE 6
Comparison of EC2 instance types running Cassandra for the Video Analytics application. *c5.large* has the best Perf/$ since the ephemeral data is small in size and is only written/read once (no parallel reads).

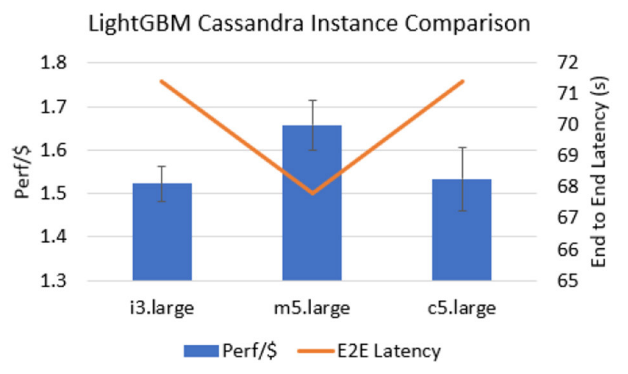

FIGURE 7
Comparison of EC2 instance types running Cassandra for the LightGBM application. *m5.large* has the best Perf/$ since the ephemeral data is large in size and the broadcast fanout of the application results in multiple parallel reads over the network (EBS).

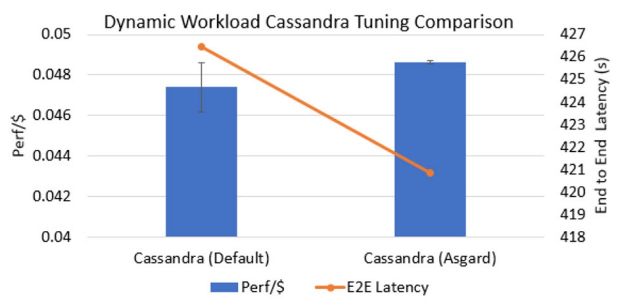

FIGURE 8
Comparison of a default Cassandra node to a node that is dynamically tuned based on the workload. Cassandra with Asgard shows a minor improvement over the default configuration since it picks the best configuration to use to optimize Perf/$. At a large scale and different request patterns, this difference can get bigger.

5. Related work

Ephemeral storage systems for serverless environments: Some prior works have identified this issue of storing ephemeral data in serverless applications and have designed storage systems to optimize on performance and cost. Pocket (Klimovic et al., 2018b) is an ephemeral data store providing a highly elastic, cost-effective, and fine-grained solution. Pocket has similar performance to ElastiCache Redis but is better optimized for cost through its intelligent storage-tier selection. Pocket is deployed to support concurrent jobs in multi-tenant use cases. It is designed to be operated primarily by a cloud provider who can offer a pay-as-you-go pricing tier to their customers since the cost of running Pocket can be amortized over different users and applications. Our work in Asgard provides a solution for a user-managed NoSQL database setup. This work can be extended to cloud datastore deployments by training a workload/performance predictor and an optimizer to generate VM configurations as defined in the design of OptimusCloud (Mahgoub et al., 2020).

Locus (Pu et al., 2019) is another serverless analytics system that combines cheap and slow storage with fast and expensive storage to reach an overall performant and cost-effective system.

Application-specific automatic cloud configurations: As shown in Figures 6, 7, it is not always trivial to pick the VM instance for an application. Other systems can be used orthogonally to the NoSQL database to provide configurations for cloud instances. CherryPick (Alipourfard et al., 2017) uses Bayesian optimization to build performance models for different applications and chooses cloud VM configurations based on these models. Selecta (Klimovic et al., 2018a) uses Latent Collaborative Filtering to predict the performance of an application with different configurations, with small training data, and uses this to rightsize cloud storage and compute resources.

NoSQL database configuration: Orthogonal to our study, it is possible to improve the raw performance of Cassandra and other key-value stores by hosting them on faster media, such as NVMe. There is a slew of works that have done this outside of the serverless context (Xia et al., 2017; Hao et al., 2020). In this paper we have shown the performance/$ operating point of Cassandra on “regular” storage media is a viable option for serverless data. Therefore, we did not feel the need to explore faster storage media in our experiments. Further, we did not need to use more sophisticated configurators for Cassandra (or NoSQL datastores in general) to show viability—there are several recent works in this space (Zhu et al., 2017; Wylot et al., 2018; Zhang et al., 2019).

6. Conclusion

Serverless functions provide an interesting set of opportunities for developers to not have to worry about the infrastructure around which their code runs. However, in more complicated workloads involving ephemeral data between different stages of a serverless application, it is not always clear what storage system to use for these intermediate datasets. In this work, we analyzed a popular NoSQL storage system, Cassandra, along with Asgard, an automatic database tuner with a cost benefit analyzer. We challenged the previous notion that NoSQL databases do not provide value for ephemeral data since they focus on durability and reliability.

Our evaluation using AWS Lambda on two popular serverless applications quantitatively shows that they have better Perf/$ than the current state-of-practice (S3), without sacrificing raw E2E latency. This happens because the NoSQL database in clustered mode can handle parallel read-write queries and the database can be hosted on the right-sized VM instance, balancing the widely varying resource requirements of the applications' data passing. We also showed that using a database tuner with workload awareness, though adding development overhead, provides benefits.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

KS: System design, running experiments, writing the paper, and generating figures. AM: System design and architecture. ZZ: Experiments against baselines. UP: Experiments against baselines. SC: Writing the paper, system design and architecture, and funding.

Funding

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS2146449 (NSF CAREER award) and funding from CNS-2038986/2038566.

Acknowledgments

The authors thank the reviewers for their insightful comments and the associate editor, Fred Douglis, for his constructive and enthusiastic feedback.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The author SC declared that they were an editorial board member of Frontiers, at the time of submission. This had no impact on the peer review process and the final decision.

Publisher's note

Author disclaimer

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., et al. (2018). "Sand: towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC)* (Boston, MA), 923–935.

Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. (2017). "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th {USENIX} Symposium on Networked Systems Design and Implementation* ({*NSDI*} *17)* (Boston, MA), 469–482.

Amazon (2021a). *Configuring Functions in the AWS Lambda Console*. Available online at: https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html (accessed December 19, 2022).

Amazon (2021b). *Configuring Functions in the AWS Lambda Console*. Available online at: https://aws.amazon.com/lambda/features/ (accessed December 19, 2022).

Amazon (2021c). *S3*. Available online at: https://aws.amazon.com/s3/ (accessed December 19, 2022).

Amazon (2021d). *S3 FAQs*. Available online at: https://aws.amazon.com/s3/faqs/ (accessed December 19, 2022).

Amazon (2022). *Choosing Your Node Size*. Available online at: https://docs.amazonaws.cn/en_us/AmazonElastiCache/latest/red-ug/nodes-select-size.html (accessed December 19, 2022).

Amazon (2023a). *Amazon States Language*. Available online at: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html (accessed December 19, 2022).

Amazon (2023b). *AWS S3 Consistency*. Available online at: https://aws.amazon.com/s3/consistency/ (accessed December 19, 2022).

Amazon (2023c). *Understanding Redis Replication*. Available online at: https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/Replication.Redis.Groups.html (accessed December 19, 2022).

Ao, L., Izhikevich, L., Voelker, G. M., and Porter, G. (2018). "Sprocket: a serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY), 263–274.

Bhasi, V. M., Gunasekaran, J. R., Thinakaran, P., Mishra, C. S., Kandemir, M. T., and Das, C. (2021). "Kraken: adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY), 153–167.

Carreira, J., Fonseca, P., Tumanov, A., Zhang, A., and Katz, R. (2018). "A case for serverless machine learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS, Vol. 2018* (Montreal, QC).

Cassandra (2021). *Apache Cassandra*. Available online at: https://cassandra.apache.org/ (accessed December 19, 2022).

Cassandra, A. (2022). *Compaction (Cassandra Documentation)*. Available online at: https://cassandra.apache.org/doc/4.1/cassandra/operating/compaction/index.html (accessed December 19, 2022).

Curino, C., Jones, E. P. C., Zhang, Y., and Madden, S. R. (2010). *Schism: A Workload-Driven Approach to Database Replication and Partitioning*. New York, NY: VLDB Endowment.

DataStax (2022). *The Write Path to Compaction*. Available online at: https://docs.datastax.com/en/cassandra-oss/2.1/cassandra/dml/dml_write_path_c. (accessed December 19, 2022).

Daw, N., Bellur, U., and Kulkarni, P. (2020). "Xanadu: mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference* (New York, NY), 356–370.

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Sign. Process. Mag.* 29, 141–142. doi: 10.1109/MSP.2012.2211477

Duan, S., Thummala, V., and Babu, S. (2009). Tuning database configuration parameters with ituned. *Proc. VLDB Endow.* 2, 1246–1257. doi: 10.14778/1687627.1687767

Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., et al. (2017). "Encoding, fast and slow: low-latency video processing using thousands of tiny threads," in *14th {USENIX} Symposium on Networked Systems Design and Implementation {NSDI} 17)* (Boston, MA), 363–376.

Gilad, E., Bortnikov, E., Braginsky, A., Gottesman, Y., Hillel, E., Keidar, I., et al. (2020). "Evendb: optimizing key-value storage for spatial locality," in *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY), 1–16.

Google (2021). *Google Cloud Functions*. Available online at: https://cloud.google.com/functions/pricing (accessed December 19, 2022).

Grother, P. J. (1995). *Nist Special Database 19 Handprinted Forms and Characters Database*. Gaithersburg, MD: National Institute of Standards and Technology.

Hao, M., Toksoz, L., Li, N., Halim, E. E., Hoffmann, H., and Gunawi, H. S. (2020). "Linnos: predictability on unpredictable flash storage with a light neural network," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (Boston, MA), 173–190.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., et al. (2019). *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* EECS Department, University of California, Berkeley, CA, United States. Available online at: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

Klimovic, A., Litz, H., and Kozyrakis, C. (2018a). "Selecta: heterogeneous cloud storage configuration for data analytics," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA), 759–773.

Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. (2018b). "Pocket: elastic ephemeral storage for serverless analytics," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (Boston, MA), 427–444.

Mahgoub, A., Medoff, A. M., Kumar, R., Mitra, S., Klimovic, A., Chaterji, S., et al. (2020). "{*OPTIMUSCLOUD*}: heterogeneous configuration optimization for distributed databases in the cloud," in *2020 {USENIX} Annual Technical Conference USENIX ATC 20)* (Boston, MA), 189–203.

Mahgoub, A., Shankar, K., Mitra, S., Klimovic, A., Chaterji, S., and Bagchi, S. (2021). "{*SONIC*}: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (Boston, MA), 285–301.

Mahgoub, A., Wood, P., Ganesh, S., Mitra, S., Gerlach, W., Harrison, T., et al. (2017). "Rafiki: a middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (New York, NY), 28–40.

Mahgoub, A., Wood, P., Medoff, A., Mitra, S., Meyer, F., Chaterji, S., et al. (2019). "{*SOPHIA*}: Online reconfiguration of clustered nosql databases for time-varying workloads," in *2019 {USENIX} Annual Technical Conference USENIX ATC 19* (Boston, MA), 223–240.

Mahgoub, A., Yi, E. B., Shankar, K., Elnikety, S., Chaterji, S., and Bagchi, S. (2022a). "{*ORION*} and the three rights: Sizing, bundling, and prewarming for serverless {*DAGs*}," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Boston, MA), 303–320.

Mahgoub, A., Yi, E. B., Shankar, K., Minocha, E., Elnikety, S., Bagchi, S., et al. (2022b). Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proc. ACM Meas. Anal. Comp. Syst.* 6, 1–28. doi: 10.1145/3530892

Microsoft (2021). *Azure Functions*. Available online at: https://azure.microsoft.com/en-us/pricing/details/functions/ (accessed December 19, 2022).

MXNet (2021). *Using Pre-trained Deep Learning Models in MXNet*. Available online at: https://mxnet.apache.org/api/python/docs/tutorials/packages/gluon/image/pretrained_models.html (accessed December 19, 2022).

Pu, Q., Venkataraman, S., and Stoica, I. (2019). "Shuffling, fast and slow: scalable analytics on serverless infrastructure," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (Boston, MA), 193–206.

Rausch, T., Hummer, W., Muthusamy, V., Rashed, A., and Dustdar, S. (2019). "Towards a serverless platform for edge {*AI*}," in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)* (Renton, WA).

Redis (2021). *Redis*. Available online at: https://redis.io/ (accessed December 19, 2022).

Röger, H., and Mayer, R. (2019). A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comp. Surv.* 52, 1–37. doi: 10.1145/3303849

Scylla (2018). *Scylladb Datasheet*. Available online at: https://www.scylladb.com/wp-content/uploads/datasheet_why-scylladb.pdf (accessed December 19, 2022).

Scylla DB (2021). *Scylladb: The Real-Time Big Data Database*. Available online at: https://www.scylladb.com/ (accessed December 19, 2022).

Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., et al. (2020). "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (Boston, MA), 205–218.

Shankar, K., Wang, P., Xu, R., Mahgoub, A., and Chaterji, S. (2020). "Janus: benchmarking commercial and open-source cloud and edge platforms for object and anomaly detection workloads," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)* (New York, NY), 590–599.

Sullivan, D. G., Seltzer, M. I., and Pfeffer, A. (2004). Using probabilistic reasoning to automate software tuning. *SIGMETRICS Perform. Eval. Rev.* 32, 404–405. doi: 10.1145/1005686.1005739

Tran, D. N., Huynh, P. C., Tay, Y. C., and Tung, A. K. (2008). A new approach to dynamic self-tuning of database buffers. *ACM Transact. Stor.* 4, 1–25. doi: 10.1145/1353452.1353455

Van Aken, D., Pavlo, A., Gordon, G. J., and Zhang, B. (2017). "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY), 1009–1024.

Wylot, M., Hauswirth, M., Cudré-Mauroux, P., and Sakr, S. (2018). Rdf data storage and query processing schemes: a survey. *ACM Comp. Surv.* 51, 1–36. doi: 10.1145/3177850

Xia, F., Jiang, D., Xiong, J., and Sun, N. (2017). "Hikv: a hybrid index key-value store for dram-nvm memory systems," in *USENIX Annual Technical Conference (USENIX ATC)* (Santa Clara, CA), 349–362.

Zafar, R., Yafi, E., Zuhairi, M. F., and Dao, H. (2016). "Big data: the nosql and rdbms review," in *2016 International Conference on Information and Communication Technology (ICICTM)* (Kuala Lumpur: IEEE), 120–126.

Zhang, H., Tang, Y., Khandelwal, A., Chen, J., and Stoica, I. (2021). "Caerus:{*NIMBLE*} task scheduling for serverless analytics," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Boston, MA), 653–669.

Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., et al. (2019). "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data* (New York, NY), 415–432.

Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., et al. (2017). "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY), 338–350.