IndiLog: Bridging Scalability and Performance in Stateful Serverless Computing with Shared Logs

Maximilian Wiesholler
TU Munich/Huawei Munich Research Center

Florin Dinu
Huawei Munich Research Center

Javier Picorel
Huawei Munich Research Center

Pramod Bhatotia
TU Munich

Abstract

State management has long been a challenge for serverless applications. Owing to their failure resilience and consistency guarantees, distributed shared logs have been recently proposed as a promising storage substrate enabling stateful serverless applications. We show that, unfortunately, state-of-the-art sacrifices compute tier scalability for log access performance, a particularly undesirable exchange for the dynamic serverless environment. The culprit is the log indexing architecture, namely relying on complete local indexes colocated with serverless functions. This design prevents efficient scaling and even risks out-of-memory errors.

IndiLog is a novel distributed indexing architecture enabling stateful serverless applications to efficiently access a distributed shared log for state management without impeding compute tier scalability. IndiLog uses a combination of local, size-bounded indexes designed to capture the expected locality patterns alongside a sharded and balanced index tier which tackles the challenges of supporting log sub-streams and bounded reads. IndiLog bests or matches Boki, a state-of-the-art distributed shared log, over various index hit rates, workload concurrency and compute tier scaling sizes.

CCS Concepts

• **Information systems** → **Distributed storage**; • **Computer systems organization** → **Cloud computing**.

Work done during Maximilian's internship at Huawei Munich.

SYSTOR '24, September 23–24, 2024, Virtual, Israel

ACM ISBN 979-8-4007-1181-7/24/09
https://doi.org/10.1145/3688351.3689159

Keywords

Serverless computing, distributed shared log

1 Introduction

Serverless state management has long been complicated by the ephemeral nature of serverless functions [8, 10, 16]. Distributed shared logs have been recently recognized as a promising state management substrate for serverless [7] because they transparently provide failure resilience and strong consistency guarantees, freeing serverless frameworks from dealing with this complexity.

A distributed shared log [2, 3, 7] is an ordered sequence of records distributed across several storage nodes. Such logs benefit from storage disaggregation [17] where the storage nodes are separated from the compute nodes for cost and manageability advantages. Internally, a distributed shared log uses an ordering tier [5] for assigning unique sequence numbers (SQNs) to records and a storage tier for storing the records. Outside of the log, a compute tier runs serverless functions which access the log via a simple API composed of append, read and trim calls. To locate the storage server storing a specific record, an index structure is internally used.

To fully realize the serverless promise of elasticity when leveraging a distributed shared log for serverless state management, two requirements are crucial. First, for performance, functions needs to access the log efficiently, especially since many are short-lived [16]. Second, the scalability of the compute tier should not be hindered by the log accesses because efficient dynamic scaling is central to serverless ability to transparently execute large bursts of functions [9].

We show that, unfortunately, the approach used by the state-of-the-art to provide efficient log access severely limits its compute tier scalability. Specifically, state-of-the-art relies on complete log indexes stored solely on the compute tier nodes [7]. When scaling the compute tier, this design leaves serverless applications with three sub-optimal options: slow log accesses, slow function start-up times or wasted resources. To explain, consider N compute nodes each storing a local, complete copy of the log index (partial local indexes have the same problems). A sudden burst of functions necessitates M extra compute nodes which lack an index copy.

M could be large relative to N as serverless bursts can be sizeable [9]. The first sub-optimal option is for the M nodes to (temporarily) query the indexes on the N nodes. However, this creates contention, slowing down the index accesses from the M nodes and even the local index accesses on the N nodes (§3). The second option is for the M nodes to delay functions until an index copy is transferred locally. This can significantly delay function completion (many functions are short-lived [16]) especially for large indexes. Also, the M nodes may only be used briefly (due to a transient burst), making the transfer of large indexes unnecessarily expensive. The third option is to proactively ensure that $N >> M$ which reduces contention when the M nodes query the remote indexes on the N nodes. This wastes resources by keeping compute nodes up solely for their index and also requires advanced workload knowledge.

A second problem with relying on complete local indexes in the compute tier is that they take considerable resources away from serverless functions. Index lookups require CPU cycles, index updates require CPU cycles and network bandwidth and storing the index requires memory. In time, large indexes can even lead to out-of-memory (OOM) crashes. We analyzed the resource utilization of the complete local indexes in Boki [7] (§3), a state-of-the-art distributed shared log for serverless. We find that index lookups have a non-trivial CPU cost. Remote index lookups from 3 4-core VMs consume an entire core on a similar VM storing the index. Also, the local index can quickly exhaust the memory on a compute node (in 3 min on a 16GB RAM VM) leading to an OOM crash. Fundamentally, complete local indexes unnecessarily couple the resources on any one compute node with the size of the entire distributed log: the maximum index size limits the total log size. Alternative approaches also have limitations. Moving the index to secondary storage can mitigate the memory usage problem but can significantly slow down log accesses, a significant problem for the short-lived serverless functions. Partial indexes can also mitigate the memory usage problem but not the CPU utilization.

This paper presents IndiLog, a distributed indexing architecture enabling serverless applications to efficiently access a distributed shared log for state management, crucially, without hindering compute tier scalability. IndiLog relies on a combination of local indexes on the compute nodes and an index tier. A local index is size-limited (thus often incomplete) and captures the locality patterns of serverless applications. The local index is optional: compute nodes can function without one to save local resources. The index tier is composed of dedicated index nodes, it is always-on and complete, i.e., it can answer any index lookup. IndiLog supports log sub-streams [3, 7, 18], a popular way to enable selective log reads as well as bounded reads [7] which use a given SQN as a bound and return the closest match to it. Together, sub-streams and bounded reads pose challenges for the index tier. Sub-streams can grow large and usually contain non-consecutive SQNs so IndiLog uses sharding to distribute non-deterministically the index for sub-streams across several index nodes. Bounded reads on such sub-streams may need index lookups that span several index tier nodes. For this, IndiLog uses a dedicated index aggregator node.

The contributions of this paper are:

- We identify and characterize the trade-off between performance and scalability made by Boki, the state-of-the-art distributed shared log for serverless.
- We present a measurement study of the resource usage and scalability implications of indexes in Boki.
- We introduce IndiLog, a novel distributed indexing architecture for distributed shared logs providing efficient log accesses and compute tier scalability.
- We integrate IndiLog with the storage and ordering tiers of a shared log.
- IndiLog achieves better or comparable performance to Boki over various index hit rates, workload concurrency and compute tier scaling sizes.

2 Background

2.1 Distributed shared logs

Overview A distributed shared log is an ordered sequence of records stored across several storage nodes. It is usually append-only, i.e., once appended data is immutable. Such logs facilitate building complex applications on top by offering useful guarantees like strong consistency, failure resilience and durability even when accessed concurrently by clients.

Component tiers The log comprises an ordering and a storage tier. The ordering tier assigns unique SQNs to records using dedicated servers called sequencers. Corfu [2] uses a single sequencer which unfortunately could be a bottleneck. Scalog [5] enables high throughput ordering using record batching, a tree of aggregators and a replicated ordering tier. The storage tier stores the data, is usually sharded, and uses data replication for failure resilience and load balancing. Data is stored in SSDs [2] or RAM.

The applications accessing the log run in a compute tier. Especially in the context of serverless workloads, there is significant benefit in using storage disaggregation [17] where the compute nodes are physically separated from the storage nodes for cost, scalability and manageability advantages.

APIs The log is accessed via a simple API composed of append, read and trim calls. The append adds a new record to the log tail and returns to the caller the corresponding, unique SQN. The read takes an SQN and returns the corresponding record, if one exists. There are two types of reads.

Scalog [5] and Corfu [2] offer point reads where the read targets a specific SQN. Boki [7] offers bounded reads, which return the next/previous record whose SQN is greater/smaller or equal to a given SQN (the lower/upper bound).

Ordering and multiplexing Several log designs [2, 5, 7] offer a single total order across all the records stored across all storage shards. Other proposals [12] provide partial ordering.

Multiple applications can write to the log concurrently, thus it is useful to create logical sub-streams in the log so that applications can selectively read data (e.g. for efficient log replays). To implement sub-streams Tango [3] uses backpointers and Boki [7] uses tags assigned to records during appends. vCorfu [18]'s materialized streams are similar.

Storing and locating records Given an SQN, a read needs to locate the storage shard storing the corresponding record. One storage node can host one or more shards. Corfu [2] uses a deterministic round-robin mapping between storage shards and SQNs. This constrained placement makes locating a record trivial, but has performance implications for both reads and writes [5]. Scalog [5] and Boki [7] avoid deterministic mappings. An SQN can be stored on any shard so an indexing mechanism is needed. Boki uses a complete log index stored in RAM and collocated with the compute nodes. Record caches can be used on the compute nodes to store recently accessed or hot records.

Applications The canonical use for a shared log involves databases which write transaction information to the log for failure resilience. After failures, the log is replayed for recovery. Several proposals have also implemented complex data structures [3] and protocols [4] on top of a shared log. Support for serverless applications can also be built on top of a shared log. Boki provides support libraries for (1) fault-tolerant workflows by adapting Beldi's techniques [20] to provide exactly-once semantics and transactions, (2) durable object storage for stateful functions by adapting Tango [3] and (3) message queues by adapting vCorfu's techniques.

2.2 Serverless requirements

When designing our indexing architecture for a distributed shared log for serverless state management we consider the strict performance requirements of serverless applications.

Fast reads Many serverless tasks are short lived [8, 16]. In the Azure Functions production workload [16], 50% of the serverless tasks run in less than 1s. Similarly, [9] describes runtimes starting from hundreds of milliseconds. This is in line with the millisecond billing granularity available today (e.g. 1ms in AWS Lambda [11]). Thus, reads and any index lookup operations that they depend on should complete as fast as possible. This also points to the benefits of servicing most index lookups locally on the nodes running the tasks.

Low start-up times Serverless tasks are known to be impacted by the start-up times (the cold start problem) of the VMs and of the frameworks they run on because the task execution times and the cold start take comparable time [15, 16]. Thus, tasks should not wait for large indexes to be replicated locally before starting as this worsens the cold start problem.

Extreme scalability Serverless applications are known for creating large, sudden bursts of tasks [9]. To accommodate the bursts, the compute tier needs to efficiently scale out. This suggests that the approach to access the shared log should in no way impact the scalability of the compute tier.

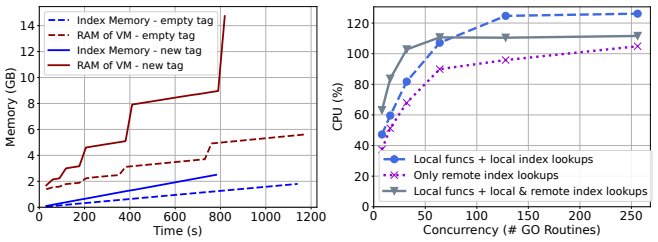
3 Motivation

This section analyzes the resource usage and compute tier scalability implications of the indexing design in Boki [7], a state-of-the-art distributed shared log. In Boki, every compute node keeps a complete index of the entire distributed shared log, in the hope that it fits in RAM thus allowing fast local lookups. The index contains tags and SQNs but no records. We use VMs (nodes) with 4 cores and 16GB RAM. The VMs run a mixed append/read workload, where an append with a new random tag completes and then the value is read back. Boki's threading model is detailed in §5. Goroutines generate append/read calls continuously. On each VM, 3 OS threads run the workload (96 Goroutines).

We find that (1) complete indexes in Boki can quickly exhaust the VM memory leading to out-of-memory (OOM) crashes, (2) index lookups can consume significant CPU cycles and (3) scaling the compute tier by allowing new compute nodes without an index to remotely query the indexes on the existing nodes significantly impacts the request latency on both types of nodes. In turn, these lessons drive the design of IndiLog (§4): IndiLog only stores partial indexes on compute nodes and a separate index tier ensures that the compute tier scalability remains unhindered.

A complete local index can quickly exhaust local RAM Figure 1a shows the RAM usage over time for one Boki node with a complete index. It is one of four Boki nodes running functions which append new records to the log. The RAM usage is measured in two ways: with OS tools (the top two dark-red lines) and inside the indexing process with knowledge of the tag and record sizes (the bottom two blue lines). The OS shows higher RAM usage because the memory allocator doubles the size of the data structures when nearly full. The dotted lines show the case when the same tag is reused. For the solid lines a new tag is used for every append.

In all cases, the RAM usage grows fast. In the worst case, when every tag is new, the system crashes with an OOM



(a) Memory usage over time **(b) CPU usage vs concurrency**

Figure 1: Resource usage for indexing in Boki.

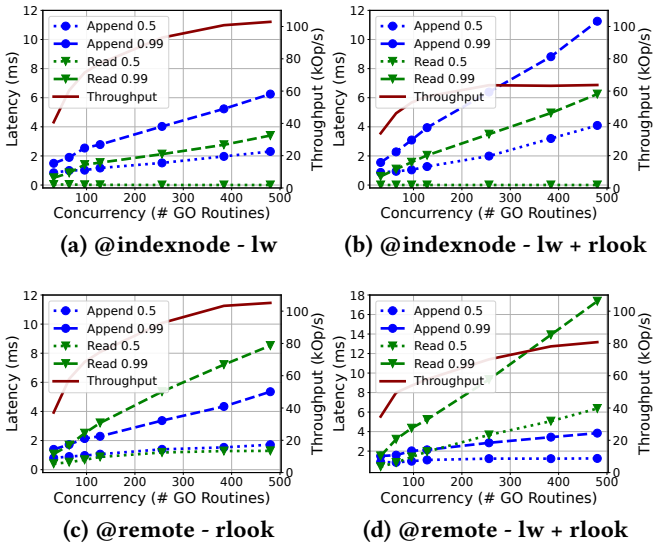
error after only 800s. At best, when one tag is reused, the RAM usage grows linearly and an OOM crash is inevitable after less than 1 hour. More RAM would only proportionally delay the inevitable OOM crash. The OOM crash tracks the OS memory usage but it would occur even based on the index-level measurement. In a real deployment the OOM would be hastened because plenty of RAM would be taken by the functions themselves. Second, the size of the index depends on the appends generated *across all compute nodes* since the index captures the entire log. We used 4 nodes so more nodes would hasten the OOM crash.

The take-away is that a complete local index is only useful for restricted workloads (few appends) or restricted duration, both of which limit applicability. Thus, an indexing design is needed that uses only partial local indexes but can still capture most of the accesses locally for good performance.

A local index uses plenty of CPU Figure 1b shows the CPU usage of a single Boki node for different concurrency levels in 3 cases: (1) only local functions and index lookups, (2) local functions and index lookups plus remote index lookups and (3) only remote index lookups. In (2) and (3) the remote index lookups are generated by 3 other nodes. We present (3) because this is the only way to scale the compute tier in Boki without waiting for entire indexes to be replicated on a new machine. 100% CPU usage means 1 core fully utilized.

The main take-away is that remote index lookups are expensive. These remote requests can use in the absence of any functions (3), one entire core on the node hosting the index (purple starred line). Comparing (1) and (2) (blue circled line vs grey triangle line) shows the impact of adding remote index lookups on top of a local workload. There is an increase in CPU usage at low concurrency but smaller than (3) (purple starred line) due to additional concurrency and locking overheads (§5). The same overheads lead to CPU utilization being slightly lower at higher concurrency.

Thus, scaling a compute tier via remote index lookups can use significant CPU resources on the nodes hosting the index. Thus, there is a need for a compute tier scaling approach that does not impact existing compute nodes.



(a) @indexnode - lw **(b) @indexnode - lw + rlook**

(c) @remote - rlook **(d) @remote - lw + rlook**

Figure 2: Impact of remote index lookups on request latency and throughput. lw = local workload on the index node, rlook = remote lookups from remote nodes.

Impact of remote index lookups on latency and throughput Figure 2 illustrates the downside of scaling the compute tier in Boki by having new nodes query indexes on existing nodes. There are 4 nodes: 1 index and 3 remote. The index node has a complete local index and may also run functions while the 3 remotes do not have a local index (emulating a compute tier scaling from 1 to 4 nodes) and send remote index lookups (only for reads, not needed for appends) to the index node. The figures show the latency (50th and 99th percentiles) for reads and appends and the throughput for each node type. On the x-axis we vary the concurrency level (simultaneously on all 4 nodes) by adding more OS threads. As per §6.1, the latencies do not include queueing time.

Figure 2a shows the latency on the index node in isolation. No remote index lookups exist in this case, only the local workload. As expected, latencies increase with concurrency and so does the throughput (up to a point). Thus, Figure 2a also points to the benefits of scaling out the compute tier to keep latencies low since a single node can only scale up so much. Figure 2b shows the same index node when it additionally serves the remote index lookups generated by the 3 remote nodes. The resulting contention significantly impacts the latencies and the throughput on the index node (Figure 2b vs 2a). The 50th percentile read latency is low because these reads are likely served from a local record cache instead of from separate storage nodes.

Figures 2c and 2d show the negative impact on one remote node's read latency caused by the contention that the remote nodes' index lookups create on the index node. Figure 2c

shows the latencies on one remote node when the index node is not running any workload. The append latency is similar to the one in Figure 2a because the append path does not include an index lookup. However, the read latency visibly increases. Figure 2d adds the workload on the index node on top of Figure 2c. The throughput and the read latencies on the remote node are visibly impacted by the contention.

Thus, there is a need to scale the compute tier without the new nodes impacting the performance of existing nodes (due to remote index lookups) and vice-versa.

4 Design

4.1 Overview

Figure 3 shows the design of IndiLog at a high level, including the four tiers (compute, ordering, index and storage) and the main control and data flow between them. More specific control flow is presented along with the append (§4.3) and the read (§4.4) paths.

The main contribution of IndiLog is the design of the indexing architecture (local indexes and index tier) and its integration with the rest of the tiers.

Innovations to the compute, ordering and storage tier are not the focus of this paper so IndiLog builds on well-established concepts. Specifically, the metalog from Boki [7] is used for failure handling, read consistency and ordering. The metalog records the log's internal state transitions and increases monotonically. Receivers of the metalog updates learn about the log's progress, remember their own metalog positions and use this information in inter-communication to prevent stale data access. As in Boki, Scalog's [5] high-throughput ordering protocol underpins the ordering tier.

API IndiLog uses a similar API to most other shared log designs, consisting of appends, reads and trim operations. As in Boki, all these operations take tags as parameters to work within logical sub-streams in the log. The reads are bounded reads as described in §2. The interaction with the indexes is transparent to the tasks because it is encapsulated in an IndiLog library running on each compute node.

High level interaction between tiers In Figure 3, serverless tasks (blue circles) running in the compute tier perform reads or appends. For reads, index lookups are needed to find which storage node stores the desired record. Index lookups may be served either locally, from the local index collocated with the task (T1.2), or remotely from the index tier when the local index lookup does not yield the relevant information (T2.1). The identified storage node is contacted either by the compute node (T1.2) or by the index tier (T2.1). The storage node then sends the record to the compute node (not illustrated). On appends (T1.1), the compute node sends data to a pre-determined shard on a pre-determined storage node.

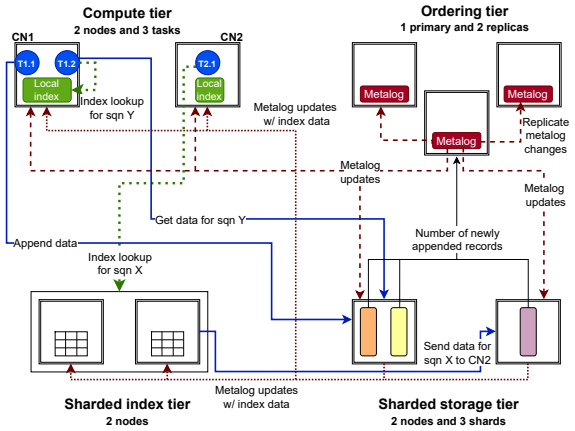


Figure 3: IndiLog's architecture along with its 4 tiers: compute, ordering, index and storage.

Periodically, the storage nodes report the number of newly appended records (since the last report) on each storage shard they own to the ordering tier using the Scalog [5] approach. The ordering tier is implemented as a primary-driven protocol. The primary sequencer is the only point of contact and it appends to the metalog which is a representation of the state of the log. The secondary sequencers replicate the metalog change. As part of the metalog update, the ordering tier assigns a contiguous range of SQNs for each storage shard. It then forwards the metalog updates (red dashed line) to the storage tier where shards derive the SQNs for each of their newly appended records and to the compute tier where pending append calls (T1.1) derive the awaited SQNs. Finally, the storage tier forwards the metalog updates with index data (red dotted line) to the compute and index tiers which can now update the indexes to point to the newly appended records.

4.2 Index data structures and properties

IndiLog uses a combination of local indexes collocated with the compute nodes alongside a separate index tier. Prior work on shared logs has either neglected indexing [5] or has assumed that complete indexes fit in each compute node's RAM [7]. We argued (§3) that the latter limits compute tier scalability and can actually quickly exhaust the RAM.

Local indexes Local indexes are optional i.e., tasks on a compute node lacking a local index can execute normally by contacting the index tier. This can allow resource-constrained nodes to participate without paying the resource cost of hosting an index. Importantly, in IndiLog compute nodes never serve remote index lookups from other compute nodes. Local indexes are size-bounded and thus often incomplete i.e., they may only have information about a subset of the SQNs.

Bounding local index size ensures that a predictable amount of memory remains available to the serverless tasks. IndiLog indexes are designed to capture the typical access and locality patterns of serverless functions [14]. Local indexes are updated based on metalog updates from the storage tier and *index the entire shared log*, not only the SQNs appended locally.

A local index handles two types of tags: an empty (default) tag and a custom function-generated tag. The empty tag is the union of all other tags and covers the entire log. All records have the empty tag, so the corresponding SQNs are consecutive. If applications wish to create a logical substream in the log, they create a custom tag. The SQNs corresponding to any one custom tag need not be consecutive.

A local index contains the following components:

- The **Suffix**. It holds the storage shards for the highest SQNs (the most **recent appends**) allocated with any tag **across the entire log**. The rationale is that recently appended SQNs [14] are likely to be accessed again. The suffix is bounded in size; the oldest entries are evicted when a threshold size is reached.
- The **Popularity Cache**. It holds the storage shards for the SQNs of the most **recent reads** accessed via any tag **on the compute node hosting this index**. Thus, these SQNs need not be consecutive. The rationale is that recently accessed SQNs are likely to be accessed again [14]. It is a size-bounded LRU cache.
- The **Tag Cache**. For each tag it stores a suffix and a last update timestamp which is the highest metalog update round that either (1) added SQNs to that tag or (2) came from the most recent read that had an index hit for that tag in the Tag Cache. When a size threshold is reached, the entries with oldest timestamps are evicted.

Index tier The index tier is sharded (for simplicity we assume one index shard per index node), always-on and complete i.e., it can definitively answer any index lookup. It is composed of one or more dedicated index nodes i.e., they only handle index operations, and one or more aggregator nodes which aggregate the best matches of the index nodes when no index node has an exact match. The index tier is designed to balance the storage space used across all index nodes. For simplicity, this paper does not consider index tier replication. However, IndiLog provides support for replicating index tier data. As a data structure, each index tier node holds a hash table from a tag to its SQNs and their corresponding storage shard in the storage tier.

4.3 Append path

Figure 4 and Algorithm 1 illustrate the append path. Tasks in the compute tier issue $append(list\langle tags\rangle, value)$ calls and get back the unique SQN assigned to that appended record. For brevity, assume a single tag per append instead of $list\langle tags\rangle$. The latter case can be derived by repeating the corresponding operations for each tag in the list.

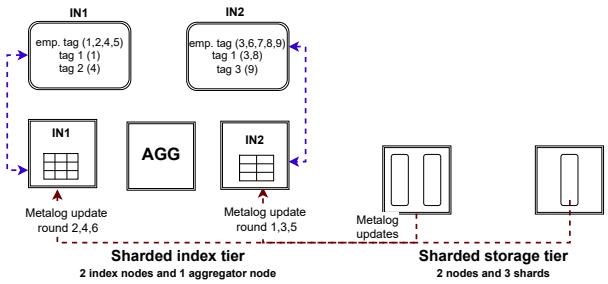


Figure 4: Index tier integration on the append path.

All tasks on one compute node append to a single storage shard and no other compute node appends to that shard. This is necessary for the Scalog [5] high throughput ordering protocol as it enables compute nodes to efficiently translate local per-shard SQNs into global ones once the metalog updates are received. Storage imbalances are avoided as storage nodes can host many shards. A compute node can start another shard, on a potentially different storage node, if the first shard is sealed.

Challenges The index tier aims to balance the index data across the index nodes to avoid imbalances that can occur when some tags are often appended to. Another goal is to minimize the overhead of the metalog updates, e.g., it is best avoided to send a metalog update to all index tier nodes.

Algorithm 1 IndiLog appends: SQN = append(tag = T, val)

```
1: procedure On the compute tier
2:     Send append to the corresponding storage shard
3:     Listen to the metalog updates to get SQN
4:     Return SQN to the calling function
5:     Incorporate metalog updates in the local index
6:     Evict index items if threshold sizes reached

7: procedure On index tier node X – on recv. metalog updates
8:     if Metalog update round % nr_index_nodes == X then
9:         X incorporates update into local index
```

Index distribution over the index tier Every metalog update, a single index shard is chosen in round-robin fashion to receive the metalog updates (Alg1:7-9). In Figure 4 there are two index tier nodes so one is chosen for the odd metalog update rounds and one for the even. Since metalog updates are frequent (in Boki every 300 µs) this ensures that index data is well distributed over the index tier. This has several important implications for reads. Tags which are appended

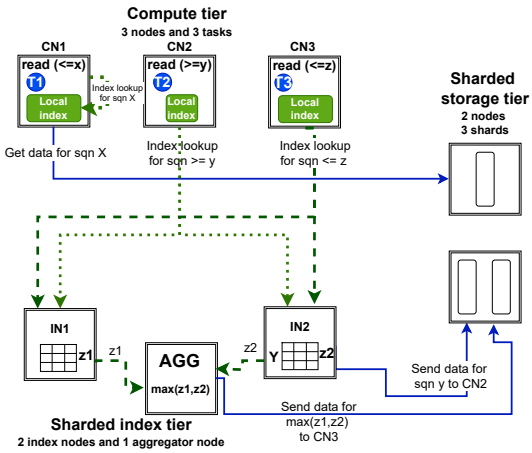


Figure 5: Integration of indexes on the read path.

to over a longer period of time appear over many metalog updates and thus are indexed over many (or all) index nodes (e.g., tag1 in Figure 4). This avoids the undesirable situation when one popular tag uses a disproportionate amount of memory on a single index node. Tags which are only briefly appended to may be indexed on as little as one index node (e.g. tag2 and tag3 in Figure 4). Any SQN can end up being indexed on any index node (non-deterministic placement).

On the compute nodes For a function, an append blocks until an SQN is returned. SQNs are derived from metalog updates from the ordering tier (Alg1:3-4). Metalog updates from the storage tier provide the information to update the local indexes (Alg1:5). Eviction occurs when the local index reaches a threshold size (Alg1:6).

4.4 Read path

Figure 5 and Algorithm 2 illustrate the read path. Tasks can issue two types of bounded reads: $read(tag, sqn <= X)$ and $read(tag, sqn >= X)$. The first returns the record with the closest SQN to X that is less than or equal to X. For clarity, Algorithm 2 only treats the first case. The second case is similar. Let OWL be the one-way network latency.

Challenges As described, the appends scatter the SQNs for a single tag, non-deterministically over the index tier in order to mitigate memory imbalances and reduce metalog update overheads. Thus, the index tier needs to involve several index nodes to answer a read. For bounded reads that return the bound, one index tier node has the exact match. However, this is not the case when a bounded read must return a closest match (e.g. when the bound does not belong to the queried tag). Since index tier nodes know nothing about each other, they cannot tell which other index node is closer so no single index tier node may be able to draw a final conclusion.

Algorithm 2 IndiLog reads: read (tag, sqn ≤ Bound)

```
1:  procedure On the compute tier
2:      if Empty Tag then
3:          Search in Suffix
4:          Search in Popularity Cache
5:      else if Custom tag then
6:          Search in Tag Cache
7:      if Local index lookup succeeded then
8:          Send read request to the identified storage shard
9:      else if Local index lookup failed then
10:         Send lookup to each index tier node tagged with the function's
    metalog update round

11: procedure On the index tier - at index nodes
12:     if Found Bound in the index then
13:         Forward read request for Bound to storage tier
14:         Send Bound to the aggregator
15:     else if Bound not found in index then
16:         Send to the aggregator closest BS, such that BS < Bound

17: procedure On the index tier - at aggregator nodes
18:     Receive BS from index node for read with unique ID RID
19:     if RID complete then
20:         Continue with next request
21:     if BS == Bound then
22:         Mark RID complete. An index node contacts storage
23:         Continue with next request
24:     if all index nodes answered for RID then
25:         REC = max (all (BS))
26:         Forward read request for REC to storage
```

Read type 1 - Exact/closest match in local index This corresponds to task T1 in Figure 5 where the local index locates the right SQN. This is an exact match when the index contains X (in Figure 5) or a closest match when the index does not contain X but rather the SQN Y preceding X for that tag (Y is in the suffix for that tag). Node CN1 then contacts the storage node hosting the SQN (Alg2:7-8). This takes $2 * OWL$ latency and 2 messages.

Read type 2 - Exact match in index tier This corresponds to task T2 in Figure 5. After an unsuccessful local index lookup, node CN2 contacts each index node (Alg2:9-10). For consistency reasons (discussed in §4.5), the request includes the function's metalog update round. In this example $IN2$ has an exact match (it indexed Y). $IN2$ immediately forwards the read request to the responsible storage node (Alg2:13) which sends the data to T2. This exchange takes $3 * OWL$ latency and the number of messages is equal to $2 * (nr\_of\_index\_tier\_nodes) + 2$. This is because the index nodes still need to message the aggregator since they do not know that another index node had an exact match. Only a single index node can have an exact match due to the way the index data is distributed over the index nodes. When the aggregator receives an exact match (Alg2:21) then it can

safely assume that the storage tier has been contacted by the index node who stored that exact match and can consider that read request completed (Alg2:22-23).

Read type 3 - Closest match in index tier This corresponds to task T3 in Figure 5. After an unsuccessful local index lookup, node CN3 contacts each index node (Alg2:9-10). Neither $IN1$ nor $IN2$ have an exact match. Neither stores Z. The index tier nodes search locally for the closest $Z\_local < Z$ and send it to the aggregator (Alg2:16) which calculates the closest value to Z (the max of all aggregated Z_local) (Alg2:24-25). The aggregator forwards the read request to the storage tier node responsible for the identified SQN (Alg1:26) which sends the data to T3. This takes $4*OWL$ latency and $2 * (nr\_of\_index\_tier\_nodes) + 2$ messages.

4.5 Other design properties

Index consistency IndiLog leverages the metalog update round as a logical consistency timestamp. Each function and each index (local or in the index tier) have such an associated timestamp. The timestamp of a function depends on its last read or append. It is either (1) that of the last index it used for a read or (2) the last metalog update round that included an SQN for an append from that function. The timestamp of an index is that of the last metalog update round which updated it. To guarantee useful properties like monotonic reads and read-your-writes consistency, the simplest approach is to disallow a function to lookup an index with an earlier timestamp than its own. Such lookups could occur if the local indexes or the index tier nodes did not get updated yet.

However, in IndiLog this condition can be safely relaxed for the index tier. It is safe for an index node to participate in some index lookups for functions with a higher timestamp if it was not meant to receive updates in the meantime (i.e., the other index nodes were selected for metalog updates). There is a limit to this flexibility. If the function has the timestamp that an index node expects next based on the round-robin schedule then the index node will temporarily block the lookup and place it in a special queue that is processed with the next metalog update.

Failure resilience For the ordering and storage tiers, IndiLog employs techniques from related work. Replication is used for both tiers and the computing framework deals with compute node failures by restarting tasks. For the index tier, the loss of local indexes due to a compute node crash is not a concern as only that node could access its local index. Index shards can be replicated. If an index lookup is impacted by an index tier failure, it is restarted by IndiLog after a timeout.

Index tier scalability We expect the index tier to need to scale up far less often than the compute tier because the index tier does not run functions and all its resources are dedicated to indexes. Still, scaling up the index tier is easy due to IndiLog's round-robin schedule for distributing metalog updates. The storage and compute nodes only need to discover a new index tier node to send it metalog updates and lookups. Aggregator nodes can also be easily added since they only keep state briefly and on a per-request basis. The index nodes only need to discover the new aggregators. A special case is when an index node is running out of memory and must be removed from the round-robin schedule for further metalog updates. This node should still participate in index lookups. Consistency is not an issue since it will not receive future updates. Adding and removing index tier nodes in IndiLog is done via a Zookeeper-like service.

5 Implementation

IndiLog is built in C++ by adding the indexing architecture on top of Boki [7]. Boki reuses the Scalog [5] high-throughput ordering protocol. The functions running in the compute tier are written in Go and use the approach from Nightcore [8].

Threading model For running the functions IndiLog uses Goroutines, a form green threads, a flow of execution managed entirely by the Go language runtime from user space. A number of Goroutines map to an OS thread (32 in our case). Each compute node in IndiLog also has I/O threads running C++ code that performs index lookups, reads or writes data and handles metalog updates. The Goroutines and I/O threads communicate via Linux pipes (FIFO queues) as in Nightcore [8]. Therefore, several appends and reads may be queued waiting for an I/O thread.

Some API calls have two distinct phases, served separately by I/O threads. For reads type 2/3 (§4.3), the first phase checks the local index and contacts the index tier. The second phase runs when data is received from the storage tier. All appends are split. The first append phase sends the data to the storage tier. The second obtains the SQN via a metalog update.

Locking Locking the index data structure is needed so that metalog updates do not impact reads. The index can change during the metalog update e.g., some arrays may be moved in memory. In Boki, the locking is coarse grained; the entire local index is locked for reads or metalog updates. For fair comparison, in IndiLog we reuse the same coarse-grained locking for both the local indexes and the index tier. For the aggregators we used a finer-grained, per-request locking.

6 Evaluation

6.1 Methodology

Setup We use cloud VMs with 16GB RAM and 4vCPUs running Ubuntu 20.04 with kernel v5.10.0. The mean latency

between VMs measured by ping is $180 \pm 40\,\mu s$. The bandwidth between two VMs measured by iperf is 2,127 Mbps.

The number of storage tier nodes is equal to the number of compute nodes and the replication factor is 1. For the ordering tier we use 3 nodes, 1 primary and 2 secondaries. Updates from the storage tier to the ordering tier and metalog updates from the ordering tier occur every 300 µs. The IndiLog index tier is sharded over 2 index nodes, uses a replication factor of 1 and 1 aggregator node. Each compute node has 4 I/O threads. Index nodes, storage nodes and sequencer nodes have 2 I/O threads. The Suffix stores up to 10^5 entries. The Popularity Cache holds up to 10^4 entries. The Tag Cache stores up to 10^6 SQNs over all tags. A single tag in the Tag Cache is limited to 10^4 SQNs. This local index configuration in IndiLog limits the size of a local index to ≈20 MB. All indexes are stored in RAM. Unless otherwise stated, we disable local record caches because we are interested in how the communication between tiers impacts performance.

Metrics We show the 50th and the 99th percentile tail latency for reads and appends. The latency measurement starts when an I/O thread first picks up a request and ends when the request returns to the Goroutine that started it. Thus, the latency does not include queueing time waiting to start the request but does include any queueing time inside the system when there is a second request phase (§5). This latency measurement allows us to understand the behavior of the system under challenging workloads and at high throughput. We show the global throughput, across all compute nodes. We also present various statistics (e.g., index hit rates) and breakdowns (e.g., different types of reads).

Workload Our workloads are composed of a mix of reads and appends. There is a balanced workload (50% reads and 50% appends) and a read-heavy workload (95% reads and 5% appends). The workloads run the functions with either high (15 OS threads/node) or low concurrency (4 OS threads/node). In order to stress the system, our functions do not perform additional computation and just continuously serve appends and reads. Generally, both appends and reads stress the system via network communication and they require locking for correctness. In addition, appends stress the system via metalog updates and reads via index lookups.

We use the empty and custom tags as described in §4.2. Record sizes are 1KB. To choose an SQN to read from a tag, our workloads use a mix of accesses (e.g. the first SQN, the last, etc). We describe this alongside each experiment. Similarly, a tag can be globally new (across all compute nodes) or can be re-used locally or globally. We also mix these.

Competitors We compare IndiLog against Boki [7], the state-of-the-art distributed shared log for serverless which uses complete local indexes. We purposely avoid the OOM errors that affect Boki (§3) by running the experiments for a shorter period of time. Boki cannot dynamically scale the compute tier so, for a fair comparison, we start Boki in the final scaled-out configuration. As in §3, we distinguish between Boki-hybrid nodes (running workload and hosting an index) and Boki-no-index nodes (the added nodes running a workload but not hosting an index and thus needing remote index lookups on Boki-hybrid nodes).

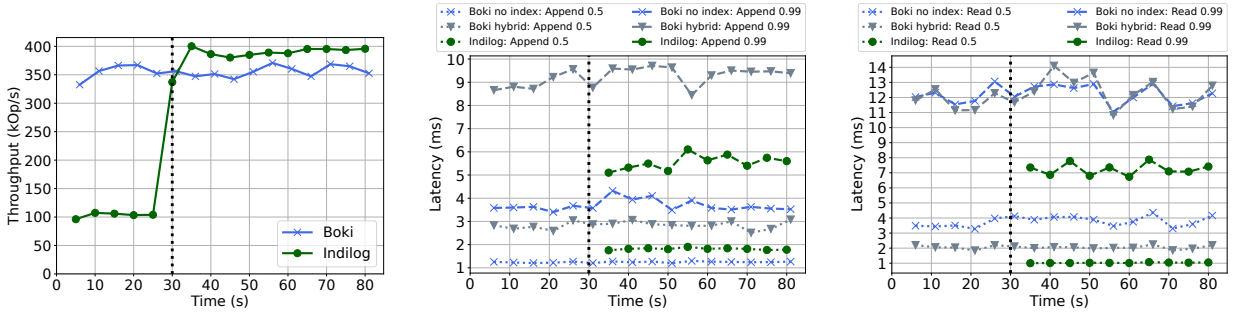
Highlights We find that IndiLog provides better or comparable performance to Boki over a range of scenarios including (1) either high or low local index hit rates in IndiLog, (2) lower or higher workload concurrency level on the compute nodes and (3) varying the scaling size i.e. the number of compute nodes dynamically joining the compute tier.

6.2 Scaling the compute tier - high hit ratio

Figure 6 shows the impact of scaling the compute tier from 1 to 4 nodes. In Boki, the 3 new nodes query the index hosted on the first node. In IndiLog, the 3 new nodes, each hosting a local index, join at second 30. This shows the envisioned case for IndiLog in which the local indexes catch most of the lookups but a small portion still goes to the index tier. The hit rate in the local indexes for IndiLog across all 4 nodes is 87%. The local hit rate for Boki is, as expected, 100%. Essentially this experiment puts in balance (1) for Boki the increased overhead on the Boki-hybrid node resulting from the contention caused by the newly added nodes and (2) for IndiLog the penalty of going to the index tier for some of the reads. Note that improving on Boki by a large margin is not the ultimate goal because in practice Boki is likely to OOM. Rather, IndiLog strives to obtain better or comparable performance without the risk of OOM.

For this experiment we use the balanced workload. Each Goroutine runs a loop where one append follows one read. The appends use equally the empty tag or a custom tag. The reads are related to the last appended tag but may or may not be related to the SQN of the last append. For an empty tag we read with equal probability (1) the appended value, (2) a popular SQN, (3) from the suffix and (4) the current log tail. For a custom tag we read with equal probability (1) the appended value, (2,3) left/right of it, (4) from the head of the log and (5) from the tail of the log.

Figure 6a shows the throughput across all 4 nodes averaged every 5s. IndiLog dynamically scales well from 110 KOp/s to 390 KOp/s and the 4 nodes have comparable throughput. In contrast, Boki achieves 350 KOp/s. Its nodes have skewed throughput: 96 KOp/s for the Boki-no-index nodes and only 60 KOp/s for the Boki-hybrid node due to the contention. Figures 6b and 6c shows median and tail (p99) latencies for appends and reads. As expected, the tail latencies

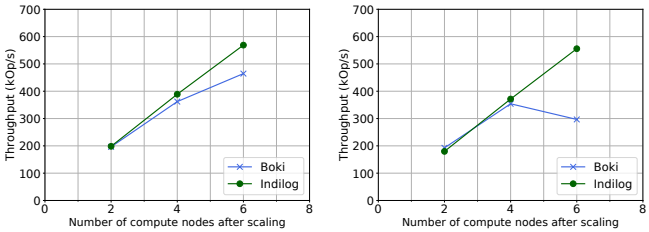


(a) Throughput **(b) Append latencies** **(c) Read latencies**

Figure 6: Throughput and request latencies (y-axis) when scaling the compute tier (x-axis = 30s) from 1 to 4 nodes. High local cache hit ratios in IndiLog. Boki cannot scale dynamically, it starts with 4 nodes. Balanced workload.

show a larger variation. The append latencies are significantly higher for the Boki-hybrid node because of the contention. In contrast, the append latencies are the best for the Boki-no-index node because it does not have a local index and thus does not pay the overhead of managing it. The reads on the Boki-hybrid node are faster than for Boki-no-index because the latter requires a remote index lookup. Finally, IndiLog reads show the best latency as IndiLog is not impacted by contention like on the Boki-hybrid node.

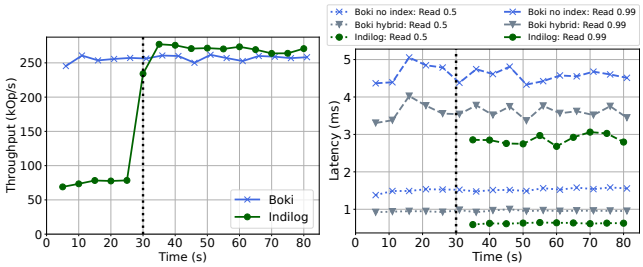
Impact of varying the scaling size Figure 7 varies the number of nodes added during scaling. We use both the balanced and the read-heavy workload. $X = 4$ in Figure 7a corresponds to Figure 6. IndiLog scales well, almost linearly. In contrast, Boki behaves increasingly worse as the size of the scaling increases ($X = 6$) due to increased pressure on the Boki-hybrid node, especially for the read-heavy workload.



(a) 50/50 Read/Append **(b) 95/5 Read/Append**

Figure 7: Varying the scaling size. The number of compute nodes grows from 1 to 2/4/6 by adding 1/3/5 nodes.

Impact of lower concurrency Figure 8 shows the lower concurrency workload. This benefits Boki by lowering the contention on the Boki-hybrid node but also reduces the overheads that IndiLog nodes have to maintain their local index. The workload change is best seen by comparing the throughput in Figure 8a and 6a. The former is significantly lower due to lower concurrency. Figure 8b shows read latencies. IndiLog still comes out on top. Compared to Figure 6c, the less intensive workload shows much lower latencies. We omit append latencies but they show a similar pattern.



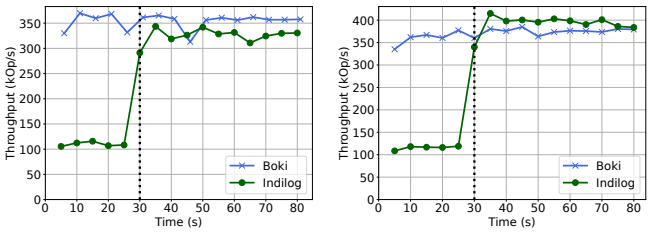
(a) Throughput **(b) Read latencies**

Figure 8: Throughput and latency. Lower concurrency.

6.3 Scaling the compute tier - low hit ratio

Next, we show a difficult case for IndiLog when most local index lookups fail and need to be serviced by the index tier. Still, IndiLog matches or bests Boki. Figure 9 shows the throughput for the balanced and read-heavy workloads. Here, the local index hit ratio in IndiLog is 20%. To obtain low hit ratios we use only custom new tags. 80% of the reads are from the head of the log and 20% read the last appended value. This yields 80% of type 3 reads (§4.4), the most expensive reads in IndiLog because they are handled by the aggregator.

Figure 9a shows a lower throughput than Figure 9b for both Boki and IndiLog because the workload has a higher ratio of appends which are the more expensive operation. Despite the low index hit ratio, IndiLog shows higher throughput than Boki in Figure 9b. In Figure 9a, Boki has a slight edge because only the Boki-hybrid node pays the overhead



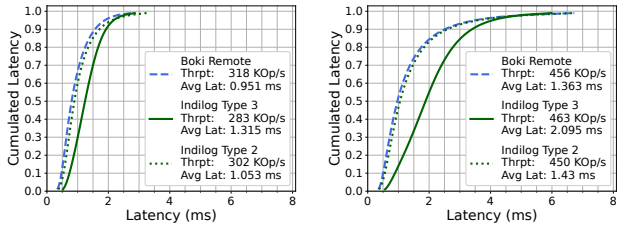
(a) 50/50 Read/Append (b) 95/5 Read/Append

Figure 9: Throughput. Scaling the compute tier (x=30s) from 1 to 4 nodes. Low local cache hit ratios in IndiLog.

of maintaining the local index via metalog updates. Instead, in IndiLog, all 4 nodes pay this overhead.

6.4 Breakdown of read latencies

Next, we single out the performance of the different types of IndiLog reads. IndiLog has 3 types of reads (§4.4) but of particular interest are the reads not present in Boki i.e. the type 2 and 3 reads served by the index tier. To single them out we use the read-heavy workload and remove the local indexes from the IndiLog nodes so all reads go to the index tier. For comparison, since IndiLog uses 2 index nodes, we give Boki 2 nodes with complete indexes which do not run functions. 4 other Boki-no-index nodes send requests.



(a) Low concurrency (b) High concurrency

Figure 10: IndiLog latency for the reads that involve index tier lookups.

Figure 10 shows the latency CDFs for both high and low concurrency. The type 2 reads in IndiLog show a very similar performance to those in Boki in both cases. The type 3 reads are, as expected, slower because they go to the aggregator node. The high concurrency increases tail latencies and also the gap between the type 2 and type 3 reads.

6.5 Scalability of the index tier

Scaling the index tier Next, we increase the number of index nodes and the number of aggregators. Table 1 shows the configurations and the resulting throughput. Scaling the index tier is mostly relevant for type 3 reads where the aggregator(s) must collect best matches from all other index tier nodes before forwarding the result to the storage tier. We reuse the high concurrency workload in §6.4 and issue only type 3 reads by reading an SQN beyond the log tail.

As expected, adding more shards decreases the throughput due to more messages between the compute tier and the index tier and between the index nodes and the aggregator. Increasing the number of aggregators does not influence the results much because in our setup that is not the bottleneck. The slight differences between 1 and 2 aggregators are explained by natural variations in the cloud network latency.

Index Tier	Thrpt. [kOp/s]	Index Tier	Thrpt. [kOp/s]
S:2, A:1	465	S:6, A:1	385
S:2, A:2	461	S:6, A:2	389

Table 1: Throughput when scaling the index tier. S = number of index shards. A = number of aggregators.

Benefit of aggregators We repeat the experiment with 2 index nodes but without a dedicated aggregator node to show its importance. For comparison, the aggregation is done by one of the index tier nodes randomly selected as master during the index tier lookup request. The throughput is 465.2 KOp/s (as above) with a dedicated aggregator and 440.6 KOp/s without. The median latency is 1.76 ms with a dedicated aggregator and 2.07 ms without. The trend is similar for the low concurrency workload: 282.4 KOp/s and 1.23 ms with the aggregator and 237.9 KOp/s and 1.62 ms without it. We expect the gap between the two solutions to increase for more index nodes.

6.6 Object storage workload

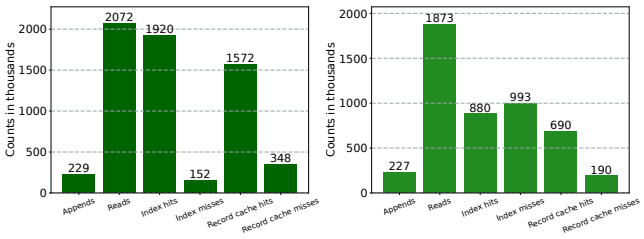
Next, using a real application, we show how the size-bounded local indexes in IndiLog and the continuous eviction from them affect performance. The take-away is that IndiLog matches Boki's performance in a real application even with local indexes far smaller than Boki's complete local indexes.

System	Thrpt. [Op/s]	System	Thrpt. [Op/s]
IndiLog	8700	Boki-Complete	8950
IndiLog-Small	8430	Boki-Remote	8381

Table 2: Throughput in the object storage workload.

We run IndiLog and Boki as the infrastructure layer of an object storage library. We reuse the BokiStore [7] library that stores objects durably on shared logs and provides transaction support. BokiStore uses only customs tags and for two reasons: (1) to find an object in the log along with its deltas so that it can apply further deltas to it and (2) to handle

transactions. We reuse Boki's Twitter clone [7] workload. We initialize it with 10,000 users and use 192 concurrent clients to trigger the functions via HTTP requests. We enable the local record cache in IndiLog and Boki.



(a) Default index size (~20 MB)(b) Small index size (~0.2 MB)

Figure 11: Aggregated statistics over all compute nodes in IndiLog with different index size configuration.

Figure 11 shows how the local index size in IndiLog affects the index hit ratio ((a): default size limits from §6.1, (b): small index (called IndiLog-SIdx)). For IndiLog-SIdx we limit the tag cache to 10^4 SQNs and a single tag to 100 SQNs before eviction. With the default limits IndiLog successfully handles 93% of all local index lookups. Even with its small index (0.2MB) almost 50% of all lookups are handled locally.

	Indilog		Indilog-SIdx		Boki Compl		Boki Remote	
Latency	50th	99th	50th	99th	50th	99th	50th	99th
Login	3.6	38.9	4.2	38.4	3.4	37.3	4.2	40.0
SeeProfile	2.6	36.6	2.7	34.0	2.7	36.0	2.8	38.8
SeeTimel.	6.4	50.8	7.9	51.3	6.0	48.5	7.9	50.3
PostTweet	8.2	49.8	10.3	52.7	7.6	48.0	9.9	51.4

Table 3: Latencies (msec) for object storage workload.

We next compare IndiLog with Boki. Each system uses 4 compute nodes to run the functions. We configure IndiLog with either the default or small local index. For Boki we also use two configurations: complete and remote. In the former, all Boki nodes store a complete index. In the latter, the 4 Boki nodes that run functions have no index and do remote index lookups to 2 other Boki nodes that do not run functions but have complete indexes. Tables 2 and 3 show the throughput and latencies for the operations in the workload. Boki-Complete performs best due to its 100% hit ratio. Its record cache serves 76% of all reads which is similar to IndiLog. Due to evictions, IndiLog-SIdx has lower throughput and higher latencies compared to IndiLog but still outperforms Boki-Remote in overall throughput.

7 Related Work

Serverless state management Other storage abstractions, apart from distributed shared logs, have been proposed for serverless state management. Neither of these works focuses on an indexing architecture for efficiently accessing the storage. At first, functions shared state via cloud object stores (e.g. Amazon S3) which is inefficient [13]. Locus [13] proposes to benefit shuffles between functions by leveraging a mix of slow but cheap storage (e.g. Amazon S3) and a small amount of memory-based fast storage to bring performance benefits while remaining cost effective. Cloudburst [17] uses Anna [19], an autoscaling key-value store for state sharing, and adds caches co-located with the functions. Cloudburst modified Anna to construct an in-storage index that maps keys to the local caches that store the key-value pairs. This index is used to propagate key updates to caches and is partitioned across server nodes. IndiLog differs in several ways. IndiLog indexes the storage while Cloudburst indexes the local caches. Importantly, Cloudburst does not analyze the scalability and performance implications of an indexing architecture. Pocket [10] is a distributed data store for the ephemeral data used by serverless functions to share state. Pocket is multi-tiered and balances elasticity and cost-effectiveness but does not tackle challenges related to an indexing architecture.

Distributed shared logs Several designs have been proposed [1–7, 12, 18] and IndiLog builds on some of these designs. However, none of these systems focuses on the design and implications of the indexing architecture which is the main contribution of IndiLog. IndiLog builds on the high-throughput ordering protocol from Scalog [5] and the meta-log from Boki [7] which itself is inspired from Delos [1]. Log sub-streams appear in Tango [3], vCorfu [18] and Boki [7]. FlexLog [6] is a very recent distributed shared log proposal. It proposes a fast storage layer based on persistent memory and a scalable ordering layer which leverages a tree of sequencer nodes. FlexLog is complementary to IndiLog because it does not tackle the challenges of the indexing architecture.

8 Conclusion

State-of-the-art systems using distributed shared logs for serverless state management sacrifice compute tier scalability for storage access performance. The culprit are the complete log indexes in the compute tier which hinder scalability while increasing the risk of OOM errors. IndiLog is a novel distributed indexing architecture for serverless applications that provides comparable or better log access performance to the state-of-the-art, crucially, without impeding compute tier scalability. IndiLog uses a combination of size-bounded, optional local indexes alongside a sharded index tier tackling challenges introduced by log sub-streams and bounded reads.

Software artifact IndiLog's code is publicly available: https://github.com/MaxWies/IndiLog

References

[1] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in delos. In *OSDI 2020*.

[2] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *NSDI 2012*.

[3] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *SOSP 2013*.

[4] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Log-structured protocols in delos. In *SOSP 2021*.

[5] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *NSDI 2020*.

[6] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. Flexlog: A shared log for stateful serverless computing. In *HPDC 2023*.

[7] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *SOSP 2021*.

[8] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS 2021*.

[9] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *SoCC 2019*.

[10] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI 2018*.

[11] Amazon AWS Lambda. New for AWS Lambda – 1ms Billing Granularity Adds Cost Savings. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-1ms-billing-granularity-adds-cost-savings/.

[12] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A partially ordered shared log. In *OSDI 2018*.

[13] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI 2019*.

[14] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *SoCC 2021*.

[15] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, page 76–84, apr 2021.

[16] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX ATC 2020*.

[17] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. In *VLDB 2020*.

[18] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vCorfu: A Cloud-Scale object store on a shared log. In *NSDI 2017*.

[19] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: A kvs for any scale. In *ICDE 2018*.

[20] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *OSDI 2020*.