DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Distributed Indexing for Dynamic Scaling in Stateful Serverless Computing with Shared Logs

Maximilian Wiesholler

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Distributed Indexing for Dynamic Scaling in Stateful Serverless Computing with Shared Logs

Verteilte Indexierung für dynamisches Skalieren in zustandbehaftetem Serverless Computing mit verteilten Logs

Author:	Maximilian Wiesholler
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dr. Florin Dinu
Submission Date:	July 15th, 2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, July 15th, 2022 Maximilian Wiesholler

Acknowledgments

I would like to thank Dr. Florin Dinu for advising me in this thesis. Thanks to his expertise I improved my research skills and got a deep understanding of distributed systems. He always took time to support me on ongoing challenges. Our conversations sharpened my mind on how to tackle a problem.

I would like to thank Prof. Pramod Bhatotia. He gave me the opportunity to do a master thesis at Huawei Research. His advice for approaching a research problem improved my academic writing and presentation skills.

I would like to thank Dr. Javier Picorel and the whole team at Huawei Research Center in Munich. It was a fantastic experience.

Finally, I would like to thank my family. They always supported me during my studies and cheered me up when work was much.

Abstract

State management has long been a challenge for serverless applications. Recently, distributed shared logs have been recognized as a promising serverless storage substrate because of their failure resilience and consistency guarantees. In the context of distributed shared logs, to fully benefit from the promise of serverless, two important requirements need to be met: the log needs to be efficiently accessed and the scalability of the compute tier must not be impeded. We show that current state-of-the-art severely limits compute tier scalability by focusing on providing efficient log access. Specifically, state-of-the-art relies exclusively on local indexes on the compute nodes running serverless functions. Unfortunately, when scaling the compute tier, this design presents a trade-off between sub-optimal options: (1) wasting resources by keeping idle compute nodes up solely for their index, (2) slow function start-up times waiting to transfer an index locally on a new compute node or (3) slow function accesses due to contention on the subset of nodes currently hosting indexes. In addition, relying solely on local indices can take away significant resources from serverless functions and risk out-of-memory errors.

This master thesis presents IndiLog, a novel distributed indexing architecture that enables serverless applications to efficiently access a distributed shared log as a storage substrate without impeding compute tier scalability. IndiLog uses a combination of local, size-bounded indexes carefully designed to capture expected locality patterns alongside a sharded index tier which balances the index sizes across index nodes and tackles the challenges of supporting log sub-streams and bounded reads. Our evaluation shows that IndiLog achieves better or comparable performance to Boki, a state-of-the-art distributed shared log, over various index hit ratios, different workload concurrency levels, read-heavy workloads, various compute tier scaling sizes and realistic workloads from applications.

Contents

1 Introduction

Serverless state management has long been complicated by the ephemeral nature of serverless functions [Sha+20; JW21b; Kli+]. As a result, serverless functions often share state via cloud storage services (e.g. Amazon S3) [PVS]. Unfortunately, this approach complicates consistency and also results in undesirable performance trade-offs [PVS; JW21a]. Recently, distributed shared logs have been recognized as a promising serverless storage substrate [JW21a] because they provide failure resilience and strong consistency guarantees. This simplifies the implementation and the concerns of the serverless frameworks.

A distributed shared log [Bal+13a; Bal+13b; JW21a] is an ordered sequence of records which are stored across several storage nodes. Such logs benefit from storage disaggregation [Sre+20] where the storage nodes are separated from the compute nodes for cost and manageability advantages. Internally, a distributed shared log uses an ordering tier [Din+20] for assigning sequence numbers to records and a storage tier for storing the records. Outside of the log, a compute tier runs serverless functions. The serverless functions 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 used.

To fully realize the serverless promises of elasticity and fine-grained resource usage when leveraging a distributed shared log for serverless state management, two requirements need to be met. First, for performance, the log needs to be accessed efficiently, especially since many functions are short-lived [Sha+20]. Second, the scalability of the compute tier should not be hindered by the log accesses because being able to efficiently and transparently execute large bursts of functions [KYK] is central to the serverless paradigm.

Unfortunately, the approach used by the state-of-the-art to provide efficient log access severely limits compute tier scalability. Specifically, state-of-the-art solely relies on log indexes stored on the compute tier nodes [JW21a]. When scaling the compute tier, this design decision leaves serverless applications with a choice between three sub-optimal options. To explain, consider N compute nodes each storing a local, complete copy of the log index (partial indexes have the same problems). A burst of functions needs to be executed next and that requires the functions to execute on M additional compute nodes which lack a copy of the index. M could be large relative to N as serverless workloads are known for their bursty nature [KYK]. The first sub-optimal option is

for the M servers to (temporarily) query indexes stored on the N nodes. However, this creates contention, slowing down the accesses from the M nodes and even the local accesses on the N nodes. The second option is for the M nodes to delay the start of the functions while a copy of the index is transferred locally. This is sub-optimal because it significantly delays the completion of functions (many functions are short-lived in practice [Sha+20]) especially since indexes can grow quite large. Also, the M nodes may only be used for a very short amount of time (due to a transient burst), making the transfer of large indexes unnecessarily expensive. The third option is to ensure ahead of time that N is large enough compared to M which can limit the negative impact of contention when the M servers query the remote indexes on the N nodes. This wastes resources by keeping compute nodes up solely for their index and also requires advanced knowledge of the workload.

A second problem with relying on local indexes in the compute tier is resource utilization: index lookups require CPU cycles, index updates require CPU cycles and network bandwidth and storing the index requires memory. If these resources are used for indexing, this limits the resources available for the serverless functions. We analyzed the resource utilization of indexes in Boki [JW21a] (§3), a state-of-the-art distributed shared log built specifically with serverless applications in mind. In Boki, each compute node stores locally a complete copy of the log index. We find that the local index can quickly exhaust the memory on a compute node (3 minutes on a 16GB RAM VM) leading to out of memory (OOM) crashes. Even in the absence of OOM errors, using complete local indexes in Boki has several disadvantages. This limits the type of workload (few appends) and limits the size of the shared log based on the maximum size of the index: the log can only grow until the index maxes out the memory of any of the compute nodes. We also find that index lookup operations can consume a lot of CPU. Remote index lookups generated by 3 4-core VMs consume an entire core on a similar VM storing the index. Storing the index on secondary storage can mitigate the memory usage problem but can significantly slow down log accesses. Partial logs can also mitigate the memory usage problem but not the CPU utilization.

This master thesis presents IndiLog , a distributed indexing architecture designed to enable serverless applications to leverage the benefits of a distributed shared log as a storage substrate. Importantly, IndiLog does not impede the scalability of the compute tier but still ensures fast log accesses. 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 is designed to capture the locality patterns of serverless applications. The local index is also optional: compute nodes can function without one to save local resources and, contrary to the state-of-the-art, they can do this without affecting other nodes.

The index tier is composed of dedicated index nodes i.e. serving index lookups is

their only purpose. It is always-on and complete, i.e., the index tier is able to answer any index lookup. IndiLog supports log sub-streams [Bal+13b; Wei+17; JW21a], a popular way to enable selective log reads. IndiLog also supports bounded reads [JW21a] which do not only target a specific sequence number but rather the closest sequence number less than or greater than the provided sequence number (which serves as a bound). Together, sub-streams and bounded reads pose challenges for the index tier. Sub-streams can grow large so IndiLog uses sharding to distribute the index for sub-streams across several index nodes. Sub-streams usually have non-consecutive sequence numbers. This, coupled with the sharding means that reads may need index lookups that span several index tier nodes. For this, IndiLog introduces a dedicated index aggregator node. Finally, we show how to incorporate IndiLog into the design of a distributed shared log that reuses established shared log design strategies (i.e. the ordering protocol from Scalog [Din+20] and the metalog from Boki [JW21a]).

Contribution

The key contributions of this thesis are:

- A measurement study of the resource usage and scalability implications of indexes in Boki, a state-of-the-art distributed shared log.
- The design of IndiLog, a novel distributed indexing architecture for distributed shared logs that provides efficient log accesses without impeding compute tier scalability.
- The integration of IndiLog with the storage and ordering tiers of a shared log.
- Our evaluation shows that IndiLog achieves better or comparable performance to Boki over various index hit ratios, different workload concurrency levels, read-heavy workloads, various compute tier scaling sizes and realistic workloads from applications.

We publish IndiLog as open source on GitHub MaxWies/IndiLog. The code of all benchmarks we present for IndiLog and Boki is also published as open source on GitHub MaxWies/IndiLog-Benchmarks.

Thesis structure

The thesis is organized into nine further chapters. In chapter 2 we provide important background about serverless computing, distributed shared logs and the design of Boki. To motivate our design decisions we explain in chapter 3 limitations of indexing

in shared logs and analyze problems of indexing in Boki in detail. In chapter 4 we state our design goals and present the design of IndiLog at a high level. After that, we explain in chapter 5 the design in detail. Chapter 6 contains the relevant information about the implementation. An evaluation, where we describe the experimental setup and present benchmarks, is given in chapter 7. In chapter 8 we summarize related work. Finally, we conclude the thesis in chapter 9 and mention possibilities for future work in chapter 10.

2 Background

2.1 Serverless Computing

Serverless computing allows the developer to focus on the application development. Configuration and management of resources for running the application is completely done by the cloud provider. Moreover, the developer is only charged for consumed resources when the application is invoked. Function-as-a-Service (FaaS) is a core concept of serverless by introducing fine-grained functions to run developer's code on a FaaS platform. Client requests or events trigger the invocation of functions and the developer pays only for the function execution. Functions are stateless and rely on additional services to manage their state. The services are offered by the cloud provider and in general billed on a per-request basis.

2.1.1 Requirements from Serverless Tasks

Serverless applications have strict performance requirements due to their promise of elasticity and fine-grained resource usage. These requirements provide guidelines for designing a service for severless state management.

Fast reads

Many serverless tasks are short lived [Sha+20; JW21b]. In [Sha+20] authors argue that 50% of the serverless tasks in the Azure Functions production workload run in less than 1 second. Similarly, [KYK] describes runtimes ranging from hundreds of milliseconds to a few minutes. This is in line with the millisecond billing granularity available in today's services (e.g. 1ms in AWS Lambda [Poc]).

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 amount of time [Sch+21; Sha+20].

Extreme scalability

Serverless applications are known for creating large bursts of tasks within a very short period of time [KYK]. To accommodate the bursts, the compute tier needs to efficiently scale out.

2.1.2 Nightcore

Nightcore [JW21b] is an open-source FaaS platform to run functions. It is composed of a frontend i.e., the API gateway, and a backend i.e., the compute nodes. The gateway receives function requests and forwards them to the compute nodes. On a compute node users-provided function code is executed in containerized RPC servers and an engine manages the execution flow.

Nightcore's system architecture aims to minimize any latency overhead in the backend: (1) internal functions calls are executed on the same compute node without going through the API gateway. (2) Nightcore's message channels between the engine and the containers are built on top of Linux pipes and are optimized for transferring small fixed-size messages. (3) Few OS threads in the engine can serve many I/O events from the function containers by using an event-driven polling mechanism. (4) Nightcore adjusts the number of concurrent functions dynamically in a compute node to get the best performance in the compute node without overloading it.

2.2 Distributed Shared Logs

A log is an ordered sequence of immutable records. Distributed logs store records across several storage nodes. A log offers clients an append-only interface. Clients that access the log can only append new records to the tail of the log. Records on the log are immutable, they cannot be changed later. In general the log is shared i.e., multiple clients append to the log and read from the log concurrently.

The main architecture of a distributed shared log has two tiers: the storage tier and the ordering tier. The storage tier backs the log and is often sharded i.e., shards of storage nodes manage partitions of the complete log. The ordering tier is responsible to assign sequence numbers to new log records. Each record in the log has a unique sequence number that identifies its position in the log.

2.2.1 API

Distributed shared logs offer the following simple API:

- `append(r)`: append record `r` to the log and return its sequence number.

- `read(s)`: return record of sequence number s.
- `subscribe(s)`: subscribe to the log. Records beginning from sequence number s are returned.
- `trim(s)`: trim the log until sequence number s.

`read(s)` implements point reads i.e., for a given sequence number the log exactly returns the record that corresponds to the sequence number. Point reads are offered by the majority of shared logs. Boki [JW21a] offers bounded reads, which return the next/previous record whose sequence number is greater/smaller than a given sequence number (which essentially serves as a lower/upper bound).

2.2.2 Consistency Guarantees

A distributed shared log that guarantees a total order among records can serve as building block for state machine replication (SMR) in distributed systems. The log records on the log encode operations i.e., state changes of the system. Machines i.e., clients, read the log to observe and to apply state changes. Since all clients read the log in the exact same order until the log's tail they can replicate the system's state. New clients can replay the log from the beginning until the tail to replicate the state as well. Clients that rejoin after a failure or after a network partition can synchronize their state by starting from the last record they observed before.

2.2.3 Ordering Strategies

The ordering tier of a distributed shared log assigns each new record on the log a unique sequence number. Two different concepts are used in state-of-the-art logs to achieve total ordering among records.

Order-first

In this concept sequence numbers are issued first and log records are stored afterwards. Clients request sequence numbers from the sequencer. They receive unique sequence numbers which they use to store the records on the log. This allows for a deterministic mapping between storage shards and sequence numbers. The sequencer is not part of the I/O path, however, if many clients concurrently request sequence numbers the sequencer can get the bottleneck. Moreover, holes in the log are generated if a client receives the sequence number but fails before appending the record. The order-first concept is used by CORFU [Bal+13a], Tango [Bal+13b] and vCorfu [Wei+17].

Persist-first

In this concept log records are stored first and the sequence numbers are issued afterwards. The storage nodes periodically send their progress to the sequencer. From these progress messages the sequencer determines the global progress among the storage nodes of each shard. The sequencer propagates the global progress of all shards to the storage nodes which can deterministically assign unique sequence numbers. The protocol achieves a higher scalable write throughput compared to the order-first concept [Din+20]. However, there is no deterministic mapping anymore and sequence numbers can belong to records of any storage shard. The persist-first concept was introduced by Scalog [Din+20]. It is reused by Boki [JW21a].

2.2.4 Streaming

Streaming tackles the challenge of log replay overheads. Clients are able to selectively read the log and jump over log entries that are not of interest.

Streams

Tango [Bal+13b] introduces sub-streams for selective reads to reconstruct object states efficiently. Sub-streams are built by storing stream identifiers and backpointers in log records. A client follows the backpointers to read a stream of log records. A log entry can store multiple backpointers of a stream which point to several previous log records of the stream. The backpointer is either a value relative to the current offset or an absolute offset. Absolute offsets locate records in the whole address space of the log but are bigger in size compared to relative offsets.

Tags

Boki [JW21a] adapts Tango's streaming concept. Instead of using backpointers it uses tags for selective reads. Each tag in Boki represents a sub-stream and maps to a (non-consecutive) list of sequence numbers.

Materialized streams

Materialized streams are introduced in vCorfu [Wei+17]. New records are stored on two different logs: on a global log and on a stream. Materialized streams are partitioned by objects i.e., each stream maps to an object. Clients profit from data locality when reading records of an object but still enjoy consistency guarantees of the global log.

The materialized stream offers random and bulk reads since all object updates are consecutively stored on disk.

2.2.5 Layered Applications

The simple API, failure resilience and consistency guarantees of distributed shared logs make them suitable to build applications on top of them. These applications can serve again as infrastructure layers for high-level applications.

The canonical application example is an object store with transaction support. The application writes object values and transaction commands for failure resilience to the log. For a Put request the application first appends a `transaction start` record. It then appends the object value. After that it appends a `commit` record. The commit is speculative because the application must read the log between the start of the transaction and the commit record to ensure that no write-conflict with another transaction exists. If there is no conflict the Put operation was successful. If there is a conflict the application appends a `transaction abort` record. For a Get request the application reads the last record of the object and returns the value.

Recently, several proposals have also implemented complex data structures [Bal+13b; Wei+17] on top of a shared log. The shared log durably persists all modifications of the data structures. A runtime allows the client to interact with the data structures and to hold the latest views in memory. It loads data structures into memory and synchronizes them by replaying the log to apply all (missing) modifications.

Implementing protocols on top of a shared log is a new use case that is presented in [Bal+b]. Machines use the log to replicate their state and to interact with other machines on the same layer. Similar to models in network communication machines of different layers are stacked and the log can be seen as the medium that transmits records between machine stacks. Machines may modify entries coming from the higher layers or extend them with their custom headers.

2.3 Boki

Boki [JW21a] is a distributed shared log built for state management of serverless functions. Libraries on top of Boki use the shared log to offer stateful services for functions, e.g., exactly-once semantics or transaction support. Boki is an extension of Nightcore [JW21b]. Compute nodes in Boki provide an API for functions to access the shared log.

2.3.1 Design

For appending new records, Boki adapts Scalog's persist-first concept: the compute nodes append new log records to storage shards in the storage tier. Each storage node periodically reports its individual local progress to the primary sequencer in the ordering tier. The sequencer periodically computes the global progress of all storage shards and shares it with all subscribers.

Boki leverages global progress messages by introducing the metalog. Boki's primary sequencer appends the global progress to the metalog. Each metalog entry is a vector of numbers, whose elements correspond to the storage shards and acknowledge their progress. The metalog is monotonically increasing and defines a total order among its entries. The primary sequencer sends the metalog to the secondaries for replication and propagates it to all subscribers. Receivers of the metalog use it to derive the new sequence numbers and progress to the next state. The total order of the metalog allows Boki to achieve multiple goals. (1) Ordering: metalog updates may arrive out-of-order at the receiver. The receiver knows which metalog number comes next and can restore the correct order. (2) Consistency: the same function can run on multiple compute nodes. However, one compute node may have not yet received the last metalog update whereas the others have. The function on the straggler must wait to avoid stale reads. (3) Failure handling: sealing the metalog pauses the state transitions in the system and allows Boki to safely reconfigure the system when a failure happened.

Boki adopts the concept of streams by introducing tags. A tag is a numeric identifier that maps to a (non-consecutive) list of sequence numbers. Log records have one or multiple tags. Log records that share the same tag belong to a sub-stream over the log. Log records of the same sub-stream can be located on different storage shards. The empty tag is applied to all log records. It builds a sub-stream over the complete log.

In Boki a sequence number can end up being stored on any shard. Therefore, for locating records, Boki uses a complete log index stored in RAM and co-located with the compute nodes. The log index contains (1) the storage shard identifiers of all issued sequence numbers and (2) all tags with their corresponding list of sequence numbers. The complete log index is built by incorporating index updates from the storage tier. The storage nodes send new index data to the compute nodes after new records have been persisted on the log.

Record caches can be used on the compute nodes to avoid the trip to the storage tier for recently accessed or hot records.

Figure 2.1 shows the architecture of Boki with the workflow for log appends. The gateway communicates with the compute tier. It forwards function requests from clients and receives the results. The compute nodes append data to the storage tier and read records from it. The primary sequencer receives the progress from the storage nodes of

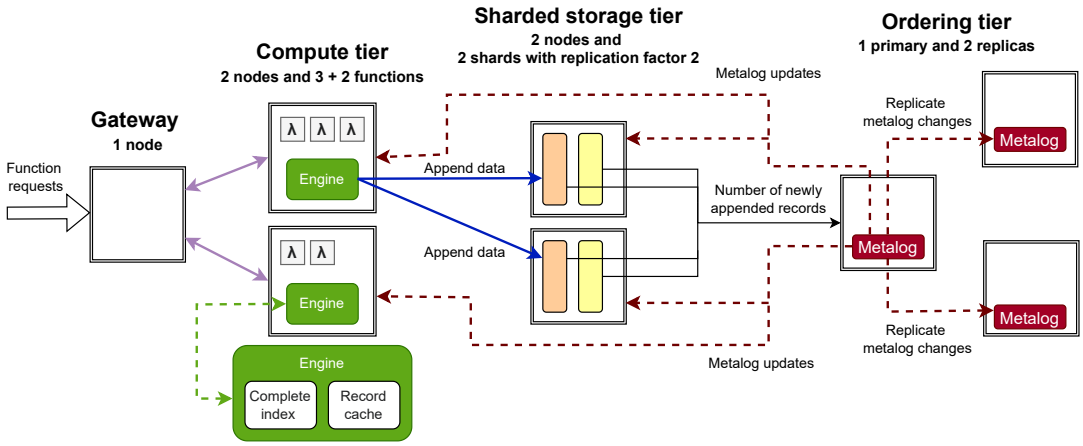


Figure 2.1: Boki's architecture including the gateway and 3 tiers: compute, storage and ordering.

the storage tier. It computes the global progress, appends it to the metalog and sends the changes to the secondary sequencers for replication. Finally, the primary sequencer propagates the metalog update to the storage tier and to the computer tier.

2.3.2 API

Boki extends the basic API of shared logs. All reads in Boki are bounded reads on sub-streams (including the sub-stream of the empty tag which covers the complete log). The API for reads covers two types and is as followed:

- `readPrevious(t, B)`: return the record in the sub-stream of tag t with the closest sequence number to bound B that is less than or equal to B
- `readNext(t, B)`: return the record in the sub-stream of tag t with the closest sequence number to bound B that is greater than or equal to B

2.3.3 Applications

Boki provides three support libraries for state management of serverless applications. All libraries use tags to realize their goals.

BokiFlow

BokiFlow offers a fault-tolerant workflow for serverless functions. It adapts Beldi's techniques [Zha+] to provide exactly-once semantics. (1) BokiFlow uses tags to recog-

nize completed steps of a workflow. Each step has a unique tag. BokiFlow appends a step as encoded log record with the step tag. After receiving the sequence number s for the appended record it immediately reads the first log record of the tag. If the sequence number of the returned record is lower than s then the step was completed before. (2) BokiFlow also uses tags to implement locks. It treats the lock identifier as a tag. If a function tries to aquire a lock BokiFlow checks if the lock is free by reading the last log record of the tag. If the log record has no holder the function can aquire the lock. For acquiring, BokiFlow appends together with the lock tag a new record that acknowledges the function identifier as holder.

BokiStore

BokiStore is adapted from Tango's techniques [Bal+13b] to build data structures on top of the log. It stores objects durably on the shared log and provides transaction support. (1) BokiStore identifies objects on the log by using their names as tags. Objects are stored as deltas on the log i.e., the state of an object is reconstructed by replaying the object's sub-stream. (2) To handle transactions BokiStore appends transaction commands with reserved tags to the log. Transaction commits are speculative i.e., BokiStore must replay the log between the start of the transaction and the commit record to detect any write conflicts.

BokiQueue

BokiQueue implements message queues by writing push and pop operations on queues as log records to the log. To determine the result of a pop the log is replayed. BokiQueue uses tags to identify queues and to handle operations (push, pop) on queues. It adapts techniques from vCorfu [Wei+17] to decompose a single queue into multiple shards such that several consumers can read from the queue without contention.

3 Motivation

In this chapter we motivate our decisions for introducing a novel distributed indexing architecture for distributed shared logs. First, we describe that indexing is neglected in state-of-the-art shared logs. After that, we analyze the indexing design in Boki [JW21a]: we show the resource usage for indexing and the implications for compute tier scalability. We support our findings with multiple benchmarks.

3.1 Neglecting of Indexing in Recent Shared Logs

Indexing allows clients to locate records on distributed shared logs. Despite its importance recent state-of-the-art logs neglect indexing. On the one hand, shared logs shift the responsibility of locating records to the client. For example, clients in Tango [Bal+13b] use backpointers in log records to build objects, however, clients either know the tail of the stream or must find it by playing the log. On the other hand, shared logs regard indexing not as an essential part of their design and use a naive solution, e.g., Scalog [Din+20] uses a remote service for indexing that may get the bottleneck.

We emphasize the importance of indexing by looking at ScalogStore [Din+20] an object store built on top of Scalog's distributed shared log. It offers a simple API for get, multi-put and test-and-multi-put operations on objects. To locate objects on the log a separate *mapping server* is used. It manages a map of object keys. Each key is mapped to a sequence number and a storage shard identifier. A sequence number corresponds to the record with the latest state of an object and the storage shard identifier locates the record in the distributed shared log. The map gets updated after the client successfully committed a new record of the object on the shared log. However, the mapping server affects the performance of the whole system. First, contacting the mapping server for index lookups increases the latency on the read path as clients must first communicate with the server to locate records. Only then they can send read requests to the storage tier. Second, the mapping server does not scale because all clients use the same single mapping server which may get the bottleneck of the system.

3.2 Implications of the Indexing Design in Boki

In Boki, every compute node maintains a complete index of the entire distributed shared log, in the hope that it will fit in RAM and thus allow fast local lookups. The index contains tags and sequence numbers but not the record values. We use VMs (nodes) with 4 cores and 16GB of RAM. The VMs run a mixed append/read workload, where an append with a new random tag is issued and completes and then the value is read back. The threading model in Boki is explained in detail in §6. In a nutshell, it is a green-threading model with N Goroutines mapping to 1 OS thread (we use N=32). Each Goroutine generates the append/read calls continuously. In the first two experiments in this section, 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 (§5): IndiLog only stores partial indexes on compute nodes and a separate index tier ensures that the compute tier scalability remains unhindered.

3.2.1 Memory Consumption of Local Indexing

A complete local index can quickly exhaust local RAM. Figure 3.1 shows the RAM usage over time for a single Boki node with a complete index. It is one of four Boki compute nodes running functions. The index is built by requests from own functions and the functions from other nodes that append new records to the log. The RAM usage is measured in two ways: with OS tools (the top two lines in dark-red) and inside the indexing process with full knowledge of the tag and record sizes (the bottom two lines in blue). The OS shows higher RAM usage because of the memory allocator that doubles the size of data structures when they become nearly full. The dotted lines show the lower bound, when the workload reuses the same tag. The solid lines show the upper bound, when a new tag is used for every request. This latter option is more expensive for the index because it needs to store every tag.

In all cases, the RAM usage grows fast. In the upper bound case, the system crashes with an OOM error after only 800 sec. In the lower bound case, the RAM usage grows linearly and an OOM crash is inevitable after less than 1 hour. A machine with more RAM would still OOM but would take proportionally longer to get there. The OOM crash tracks the OS memory usage but it would occur even based on the index-level measurement. Note that in a real deployment the RAM would run out even faster for several reasons. First, plenty of RAM is needed for the functions themselves yet in this

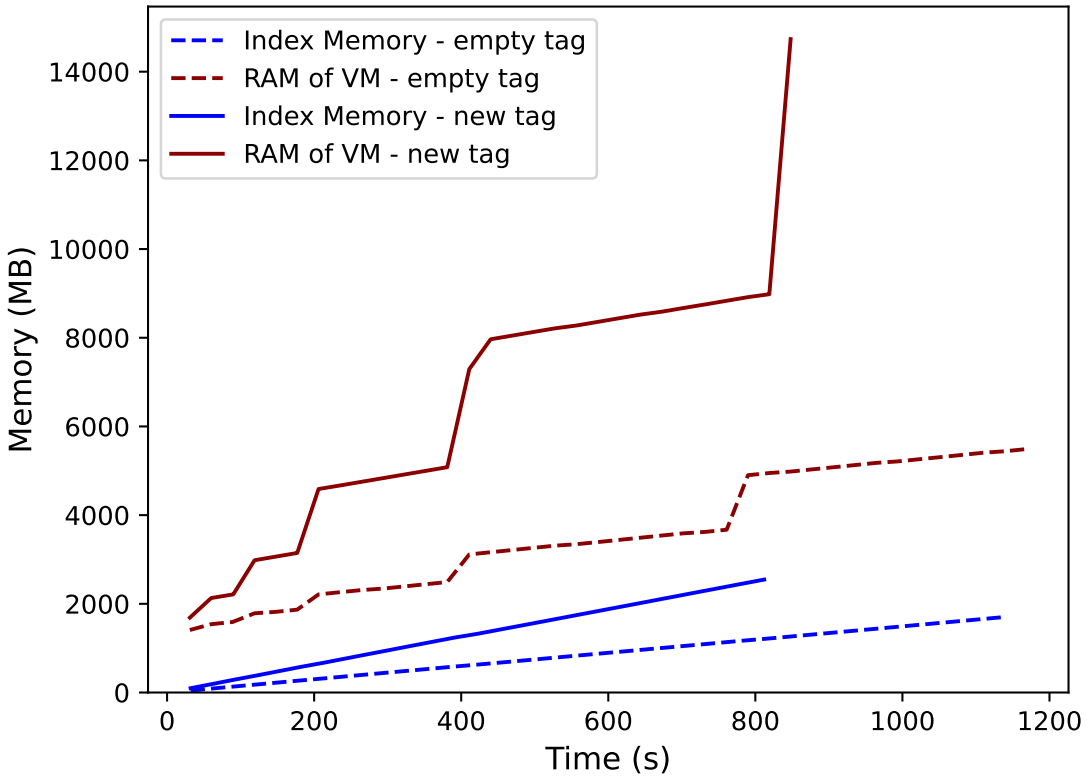


Figure 3.1: Memory usage over time for an index in Boki.

example we purposely allowed the index to take most of the RAM. Second, the size of the index depends on the appends generated *across all compute nodes* since the index captures the entire log. In this experiment 4 nodes run the workload. More nodes would hasten the OOM crash.

The take-away is that a complete index is not feasible unless the workload is very restricted (very few appends) or the shared log is available for a short time (which limits 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.

3.2.2 CPU Consumption of Local Indexing

A local index can use plenty of CPU. Figure 3.2 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

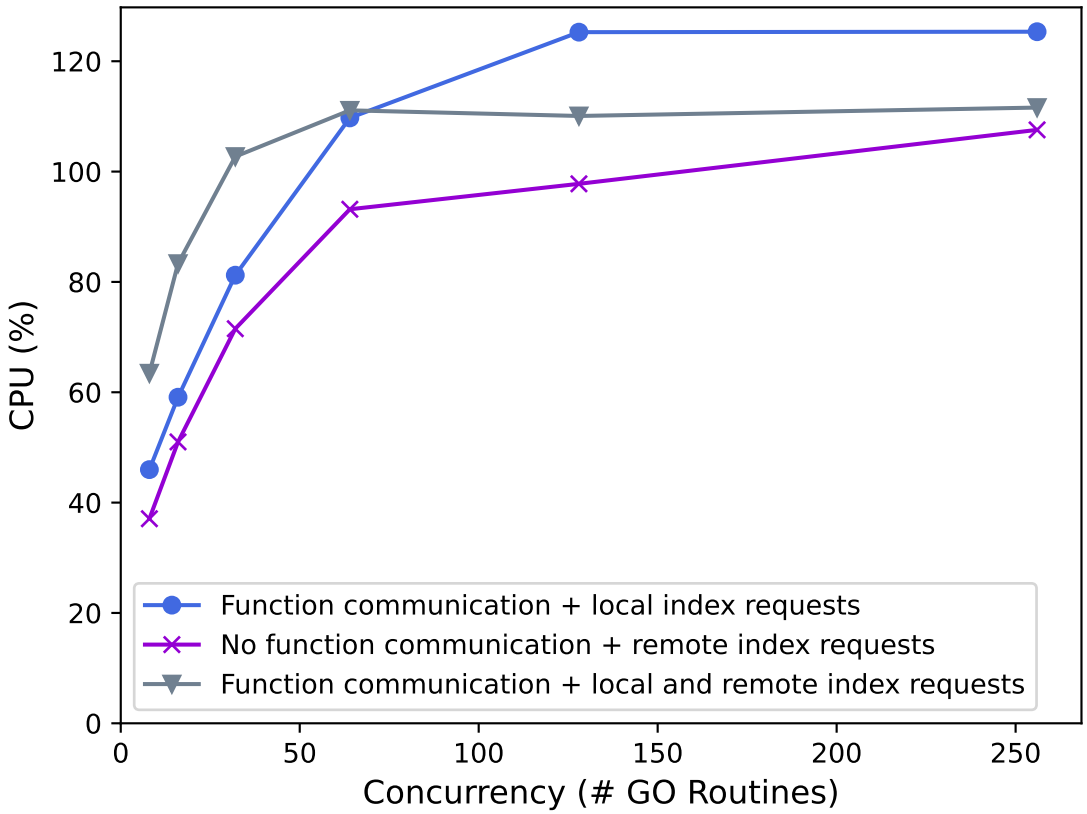
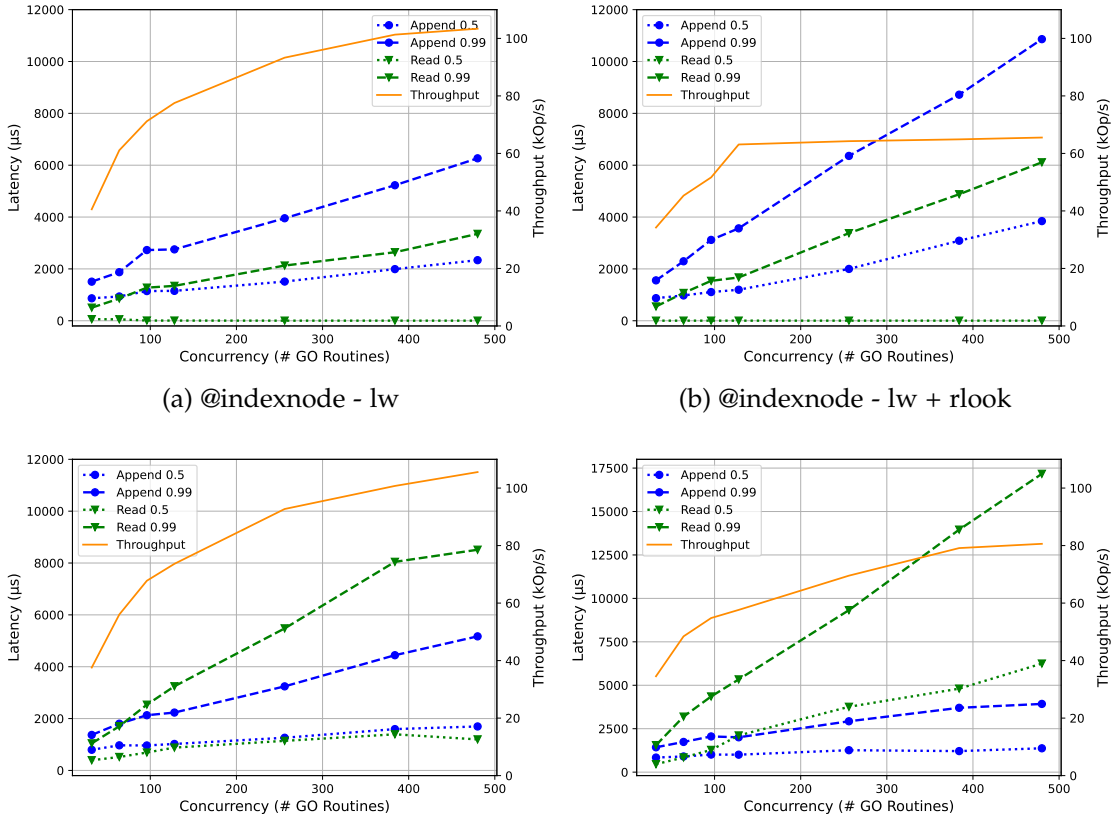


Figure 3.2: CPU usage vs concurrency in Boki.

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 utilization means one 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 (§6). The same overheads leads to CPU utilization being slightly lower at higher concurrency.

These results suggest that 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 3.3: Impact of remote index lookups on request latency and throughput. lw = local workload on the index node, rlook = remote lookups from remote nodes.

3.2.3 Impact of Remote Index Lookups

Figure 3.3 shows how the request latency is impacted on *both types of nodes* when nodes without an index send remote index lookups to a node with an index which is also running a workload. Thus, this scenario illustrates the significant downside of scaling the compute tier in Boki by allowing new nodes to query indexes on existing nodes. In this scenario there are 4 nodes. One node has a complete local index and may also run functions while the other 3 nodes do not have a local index (thus simulating a compute tier scaling from 1 to 4 nodes) and send remote index lookups (only for reads, not needed for appends) to the first node. We call the first node the index node and the other 3 remote nodes. The figures show the latency (50th and 99th percentiles) for

reads and appends as well as the throughput on a specific node. On the x-axis we vary the concurrency level by adding progressively more OS threads. The concurrency level is varied simultaneously on all nodes. As described in §7.1, these latencies are measured from the moment an append or read start being processed until they return to the user i.e., no queueing time before execution is included.

Figure 3.3a shows the latency on the index node in isolation. No remote requests 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 3.3a also points to the benefits of scaling the compute tier to keep latencies low since on a single node latencies grow with concurrency. Figure 3.3b shows the same index node when it additionally serves the remote index lookups generated by the 3 remote nodes. Despite the index lookup being an inexpensive operation in itself, the contention brought by the remote index lookups significantly impacts the latencies and the throughput on the index node (Figure 3.3b vs Figure 3.3a). In both figures, the 50th percentile read latency is much lower than the other latencies because these reads are likely to be served from a local record cache instead of from separate storage nodes.

Figures 3.3c and 3.3d show the negative impact on the remote node's request latency caused by the contention that the remote node's index lookups create on the index node. Figure 3.3c shows the latencies of the remote node's requests when the index node is not running any workload. The append latency is similar to the one in Figure 3.3a because the append path does include an index lookup. However, the read latency visibly increases due to the reads requiring a remote index lookup. Figure 3.3d adds the workload on the index node on top of Figure 3.3c. The throughput and the read latencies on the remote node are visibly impacted by the contention.

The take-away point is there is a need for an approach that scales the compute tier without the new nodes impacting the perform of existing nodes (due to remote index lookups) and vice-versa.

3.2.4 Partial Indexes

In the paper that introduces Boki it is shortly mentioned that a complete index cannot be maintained in a large-scale deployment [JW21a]. However, no further details are given regarding how the system would work if the indexes on the compute nodes are indeed not complete. We assume that in such deployment the index on a compute node is partial. It contains index data from records appended by the compute node and possibly index data of other compute nodes to satisfy the index replication policy of the shared logs. The partial index can reduce memory consumption but new problems appear.

To begin with, partial indexes can get unbalanced across the compute tier. Compute

hotspots can lead to index hotspots: if applications with append-heavy workloads run on a single compute node, then the partial index will grow faster than partial indexes on other compute nodes. With a larger index size chances are higher that the partial index must handle more index read requests. For partial indexes there exists no mechanism to mitigate hotspots.

Another implication of partial indexes is index data fragmentation. Functions of applications are allowed to be executed on any compute node. This means that index entries of an application may be located on all compute nodes. For index queries a compute node must send requests to all other compute nodes and merge the results to get the correct result. This adds load among all nodes in the compute tier.

Furthermore, partial indexes would face the following scaling problems:

- Index reads do not scale with new compute nodes.
- Compute nodes with a partial index cannot be removed if this violates the replication policy. This is especially a problem for burst scenarios: we add a compute node but cannot remove it later because the node contains index data. Even if the node has no functions to execute, it must be kept alive because of its index and thus wastes resources.
- Adding short-living compute nodes with partial indexes and replicating the indexes on long-living compute nodes induces high memory load on the long-living ones.
- Adding compute nodes with no indexes increases load on compute nodes with partial indexes.

4 Overview

Figure 4.1 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 (§5.3) and read (§5.4) paths. IndiLog reuses well-established concepts in distributed shared log design for the storage and the ordering tier. Specifically, IndiLog reuses the metalog concept from Boki [JW21a] for failures handling, read consistency and disseminating the ordering of log records. Boki itself uses concepts from Delos [Bal+a] for failure handling and uses Scalog's [Din+20] high-throughput ordering protocol for the ordering tier.

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.* Local indexes are size-limited (thus often incomplete) and optional. The index tier is complete and able to serve any index lookup. IndiLog aims to achieve the following design goals coherent with the requirements from serverless tasks (§2.1.1) :

Performance

Reads are fast in the common case as most index lookups are captured by local indexes. The index lookups on the index tier are fast as index nodes do not run functions and maintain only a shard of the complete index.

Resource efficiency

Local indexes are small in size to keep memory and CPU utilization low. Storage space for index data is balanced across all index nodes nodes of the index tier.

Scalability

The scalability of the compute tier is not impacted in any way by the design of indexing.

Functions run anywhere

Functions do not experience any restrictions where they run on the compute tier in any way by the design of indexing.

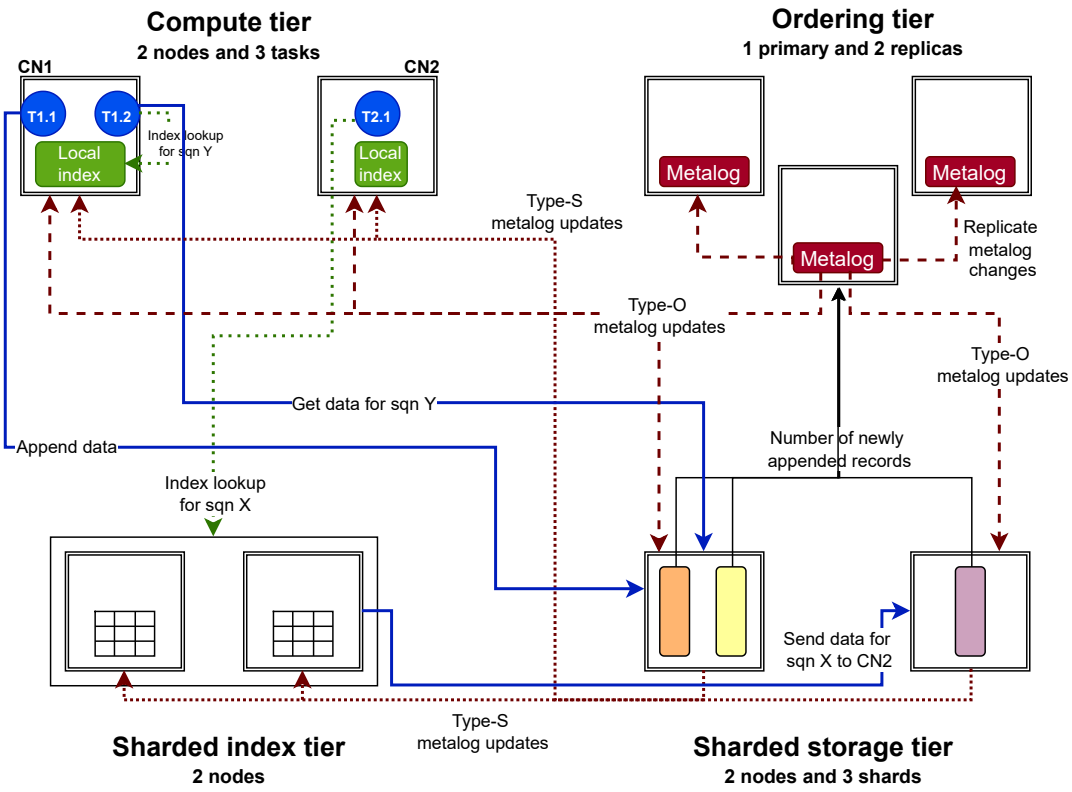


Figure 4.1: IndiLog's architecture including 4 tiers: compute, ordering, index and storage.

4.1 System Workflow

API

IndiLog uses a similar API to most other shared log designs, consisting of append, read and trim operations. As in Boki, all these operations take tags as parameters in order to work within logical sub-streams in the log. The reads are bounded reads as described in §2.3.2. The interaction with the local indexes and the index tier is transparent to the tasks because this interaction is encapsulated in an IndiLog library running on each compute node.

High level interaction between tiers

In Figure 4.1, serverless tasks (blue circles) running in the compute tier perform reads or appends to the log. 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 (e.g. task T1.2), or remotely from the index tier in case the local index lookup does not provide the relevant information (e.g. task T2.1). The desired storage node (pointed to by the index lookup) is contacted either by the compute node (after a successful local index lookup e.g. T1.2) or by the index tier (after a successful index tier lookup). The storage node then sends the record to the compute node (not illustrated). On appends (e.g. task 1.1), the data is sent to an already known storage node. Appends do not need index lookups.

To distinguish metalog updates from the ordering tier and from the storage tier, we call the former ones **type-O metalog updates** and the latter ones **type-S metalog updates**. Type-O metalog updates contain the storage shards' progresses. Type-S metalog updates are built from type-O metalog updates and contain new index data. Note that type-S metalog updates are sent by all storages nodes. However, for simplification we treat type-S metalog updates as a single message from the storage tier.

Periodically, the storage nodes send the number of newly appended records (since their last report) on each storage shard to the ordering tier using the approach in Scalog [Din+20]. The ordering tier is implemented as a primary-driven protocol. One sequencer node (the primary) is the only point of contact and it appends to the metalog. The metalog is a representation of the state of the log. The rest of the sequencers replicate the change to the metalog made by the primary. As part of the metalog update, the ordering tier assigns a contiguous range of sequence numbers for each storage shard. It then forwards the type-O metalog updates (red dashed line) to the compute tier and the storage tier. The pending append calls (T1.1) in the compute nodes use the type-O metalog updates to obtain the sequence numbers for the appended records. The storage nodes use the type-O metalog updates to derive the sequence numbers for their replicated records. Finally, the storage tier forwards the type-S metalog updates (red dotted line) to the compute and index tier which can now update the indexes to point to the newly appended records.

5 Design

IndiLog uses a combination of local indexes collocated with the compute nodes alongside a separate index tier. Prior work has either neglected indexing, focusing on other aspects of shared logs [Din+20] or has assumed that complete indexes fit in the RAM of each compute node [JW21a]. We have argued that this latter approach limits the scalability of the compute tier and shown (§3.2.1) that local indexes can quickly exhaust the RAM on a compute node.

5.1 Local Indexes

Local indexes are collocated with compute nodes. They 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 sequence numbers and thus may only be able to answer a subset of the index lookups. The size-bounded property is important because it 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 [Rom+]. Local indexes are updated based on metalog updates from the ordering and storage tier and contain information about the entire shared log, not only about the sequence numbers appended locally.

A local index makes the distinction between 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, thus the sequence numbers corresponding to the empty tag are always consecutive. When applications are interested in creating a logical sub-stream in the log, they use a custom tag. Since each record could have a different tag and the log sequence numbers are unique, the sequence numbers corresponding to custom tags need not be consecutive.

In the following we present the components of a local index in a compute node. We explain what kind of index data they hold and how they process index lookups for bounded reads.

5.1.1 Suffix

This component holds the storage shards ids for the highest sequence numbers (the most **recent appends**) allocated with any tag **across the entire log** and not only on the node hosting this index. The rationale is that recently appended sequence numbers [Rom+] are likely to be accessed again. The suffix is bounded in size; the oldest entries are evicted when a threshold size is reached.

The design of the Suffix is very compact. With less memory overhead the compute nodes in IndiLog can store a longer tail of sequence numbers and postpone the eviction of the oldest entries.

We achieve the compactness of the Suffix by making use of the batching characteristic in shared logs that apply the persist-first concept. The sequencer periodically propagates the new progress of the storage shards as a batch. In IndiLog this batch is the metalog update. The information allows all receivers to derive the sequence numbers for the latest replicated log records. To avoid any misconception the formula to derive the sequence numbers must be deterministic across all receivers. This is achieved by a predefined ascending order of the storage shards and their progress. Independently from the individual progress of a storage shard all receivers of the metalog apply the same ordering to deterministically assign the new sequence numbers. Consequently, for each metalog update all new sequence numbers of records persisted on a storage shard which comes after another storage shard in the order are greater than the new sequence numbers of the other storage shard.

In IndiLog the type-O metalog update contains the progress of the storage shards and is incorporated into the Suffix. The Suffix needs only to store the upper bounds of new sequence numbers of productive storage shards in ascending order. A sequence number that lies between two upper bounds belongs to the higher one. Productive storage shards are those shards that replicated new log records in a metalog update.

From the metalog updates the Suffix builds a chain. Each metalog update extends the chain. The key of a chain member is set by the highest upper bound of the progress in the metalog update i.e., the highest newly assigned sequence number. To save memory a chain member does not store the upper bounds of the lower storage shards. Instead it stores the distances from the original upper bounds to the key i.e., the highest upper bound. We motivate our design decision by looking at the size of a sequence number: its integer representation requires more space than the distance between two upper bounds within a metalog update. The distance between two upper bounds is always very small and a 2B integer sufficient whereas global sequence numbers in shared logs need 8B integers or even more.

Figure 5.1 shows an example sketch of the Suffix. The upper row represents the keys built by the highest derived sequence numbers from the metalog updates. The middle

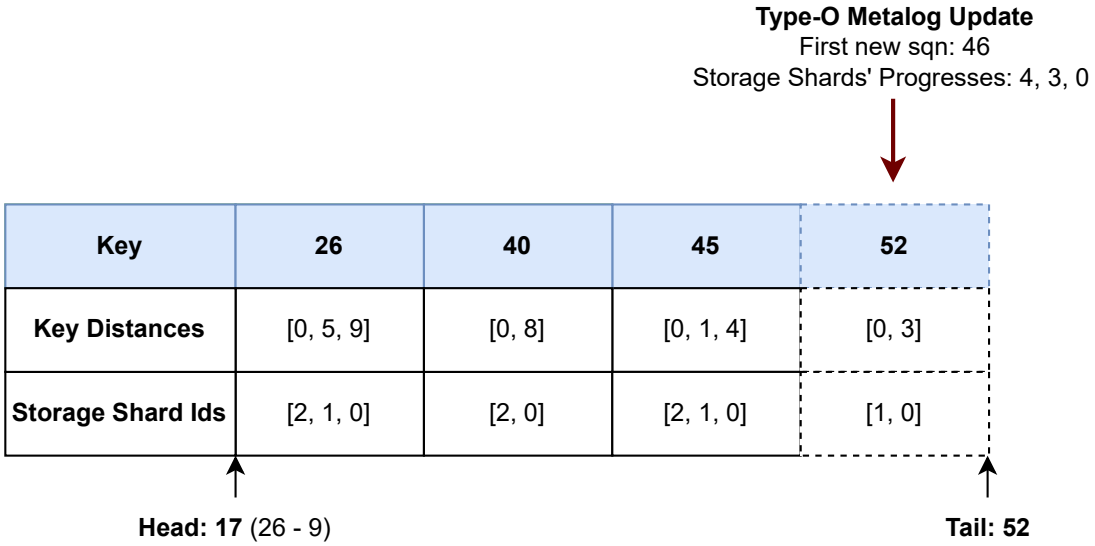


Figure 5.1: Illustration of a suffix: the suffix builds a chain from received metalog updates. Three storage shards with ids 0, 1 and 2 back the log. The new metalog update contains progress from storage shard 0 and 1 which replicated 4 and 3 new log records, respectively.

row shows the respective distances to the key `key_distances` and the lower row the corresponding storage shard ids `storage_shard_ids`.

The semantics for an index lookup is shown in Algorithm 1. It treats the case for an index lookup of type *ReadNext*: if there is a hit the index lookup returns the storage shard id of the sequence number that is equal or higher than the provided bound. Type *ReadPrevious* is similar. The core of the algorithm consists of two binary searches. The first binary search finds the chain member (Alg1:9). The second binary search finds the storage shard id within the chain member (Alg1:11-12). The lookup has $log(n) + log(m)$ complexity in time, where n is the length of the chain and m is the maximal length of a chain member. Note that m is limited by the number of existing storage shards in the system. In general the formula is dominated by the length of the chain, thus the complexity in time is $log(n)$ in most cases. An index lookup within the head and the tail is always a hit because the suffix is contiguous. An index lookup with a sequence number lower than the head is a miss (Alg1:2-4) but the lookup takes only constant time.

Comparing Suffix's space efficiency to the space requirement of a naïve design supports our design decisions. Let us assume that sequence numbers are 8B integers

Algorithm 1 Index Lookup in Suffix: readNext(Bound = B)

```
1: procedure ON THE COMPUTE TIER
2:     if B < Head then
3:         Return index miss
4:     end if
5:     if B > Tail then
6:         Return index cannot exist
7:     end if
8:     ▷ Binary-search in Chain
9:     Get member M with key k for which k is lowest upper bound of B
10:    ▷ Binary-search in key_distances of M
11:    d := k - B
12:    Get index i of entry d' for which d' is highest lower bound of d
13:    Return storage_shard_ids[i]
14: end procedure
```

and storage shard ids are 2B integers. Moreover, let us assume that we have 4 storage shards that are all productive in a metalog update and replicate 10 new log records, respectively. In a naïve design the compute node consumes $4 * 10 * (8B + 2B) = 400B$ in a sinlge metalog round. Under the assumption that distances between two upper bounds fit into 2B integers the Suffix consumes only $8B + 4 * (2B + 2B) = 24B$ per metalog round. *The Suffix consumes 94% less memory compared to a naïve design.*

5.1.2 Popularity Cache

The **Popularity Cache** holds the storage shards for the sequence numbers **less than the head of the Suffix (§5.1.1)** of the most **recent reads** accessed via any tag *on the node hosting this index*. Thus, the sequence numbers in this cache need not be consecutive. The rationale is that recently accessed sequence numbers are likely to be accessed again [Rom+]. It is implemented as a size-bounded LRU cache.

The cache gets filled with read results that were index lookup misses. The Popularity Cache is not filled with index hits from the Suffix to reduce contention. If a popular sequence number that is close to the tail of the log is first served by the Suffix, one of the later lookups is eventually an index miss when the sequence number got evicted from the Suffix. However, after a single miss all the following index lookups are captured again locally by the Popularity Cache.

The semantics for an index lookup is shown in Algorithm 2. The Popularity Cache is agnostic to bounded reads. It can only have point hits (Alg2:2) i.e., the lookup is a hit if the Popularity Cache stores the bound with the corresponding storage shard id. Lookup semantics for *ReadNext* and *ReadPrevious* are identical.

Algorithm 2 Index Lookup in Popularity Cache: readNext(Bound = B)

```
1: procedure ON THE COMPUTE TIER
2:     if B in Cache then
3:         ▷ Get and return the storage shard id of B
4:         Return Cache[B]
5:     else
6:         Return index miss
7:     end if
8: end procedure
```

5.1.3 Tag Cache

The **Tag Cache** is a map from custom tags to the following fields: `suffix`, `storage_shard_ids`, `last_access_time`.

`suffix` holds the highest sequence numbers that were derived by the appends of an tag. `storage_shard_ids` holds the storage shard identifiers of sequence numbers. `last_access_time` is a logical timestamp each tag has. An event (insert, update, read hit) that accesses the tag overwrites the timestamp with the current metalog position. We use the metalog position because it is monotonically increasing and offers a total order. A tag with a higher `last_access_time` received an event later in time than those entries with lower `last_access_time` values. When the Tag Cache reaches its size limit, the last access time is used to evict those tags completely that have not been recently had their timestamp updated.

An application may append new records to the same custom tag all the time. Therefore, the suffix of a tag is limited in size. If the suffix of a tag exceeds the limit, older sequence numbers get evicted.

IndiLog also supports an **extended Tag Cache**. It manages two more fields per tag: `min_sqn` and `complete_bit`. Compute nodes maintain an extended Tag Cache if the identification of new tags is activated in IndiLog. In IndiLog compute nodes are dynamic: they join the compute tier but may get removed later. In consequence, a compute node cannot know without additional information whether a tag was used before or is new. We present in §5.7 a solution to identify new tags.

`min_sqn` is the lowest sequence number of a tag. We store this value for two reasons: first, workloads of libraries built on top of IndiLog may lookup the minimum sequence number of a tag often, e.g., BokiFlow [JW21a] does so to offer fault-tolerant workflows (§2.3.3). Second, the tag cache needs the minimum sequence number to determine whether the tag is new. However, the combination of `suffix` and `min_sqn` is not sufficient to capture all index lookups for a new tag. The tag cache does not not know whether the tag has sequence numbers that lie between `min_sqn` and the head of `suffix`. It is a gap. Index lookups that use bounds that lie in the gap are false-negatives, if a

compute node has seen all updates for the tag and no eviction for the tag happened yet i.e., the tag is complete. To avoid false-negatives the tag cache maintains a helper bit for each tag, the `complete_bit`. If the tag is complete, it is set to 1. Index lookups for a complete tag are always hits.

Figure 5.2 sketches the Tag Cache and the extended Tag Cache. The procedure of an index lookup in the Tag Cache and in the extended Tag Cache is shown in Algorithm 3 and Algorithm 4, respectively. Both algorithms first do a check if the tag is stored in the (extended) Tag Cache (Alg3:2, Alg4:2). Tags are hashed so the check needs amortized O(1) in time. The lookup in the extended Tag Cache considers the minimum sequence number of the tag (Alg4:6) and whether the tag is complete (Alg4:11). Finally, both algorithms do a binary search over the tag's `suffix` (Alg3:13, Alg4:22) which takes $log(n)$ complexity in time where n is the length of `suffix`.

Tag	Suffix	Storage Shard Ids	Timestamp
tag 1	[4,6,7,9,10]	[1,0,1,2,0]	5
tag 2	[3,7,8,11]	[1,0,0,2]	5
tag 3	[5]	[2,2]	3

(a) Tag Cache

Tag	Min Seqnum	Suffix	Storage Shard Ids	Complete	Timestamp
tag 1	4	[6,7,9,10]	[1,0,1,2,0]	1	5
tag 2	3	[7,8,11]	[1,0,0,2]	1	5
tag 3	1	[5]	[2,2]	0	3

(b) Extended Tag Cache

Figure 5.2: Illustration of the (extended) Tag Cache. Three storage shards with ids 0, 1 and 2 back the log. The extended version additionally manages for each tag the minimum sequence number and the complete bit. Tag 3 in the extended Tag Cache is not complete because sequence numbers 1 and 2 got evicted. However, sequence number 1 remains as the minimum sequence number of tag 3.

5.2 Index Tier

The index tier is sharded (for simplicity we assume one index shard per index node from now on), always-on and complete i.e., it is able to definitively answer any index lookup. It is composed of one or more dedicated index nodes i.e., dealing with index operations is their only activity. The index tier is designed to balance the storage space used across all index nodes. For simplicity, the rest of this thesis does not consider index tier replication i.e., it assumes a replication factor of 1. However, IndiLog provides support for replicating index tier data if desired. As a data structure, each index tier node holds a map from a tag to its sequence numbers and their corresponding storage

Algorithm 3 Index Lookup in Tag Cache: readNext(Tag = t, Bound = B, Metalog position = p)

```
1: procedure On the compute tier
2:     if t not in Tag Cache then
3:         Return index miss
4:     end if
5:     ▷ Values of following tag fields (e.g. Suffix) belong to t
6:     if B < head of Suffix then
7:         Return index miss
8:     end if
9:     if B > tail of Suffix then
10:        Return index cannot exist
11:    end if
12:    ▷ Binary-search in Suffix
13:    Get index i of sequence number s for which s is lowest upper bound of B
14:    Set timestamp to p
15:    Return storage_shard_id[i]
16: end procedure
```

Algorithm 4 Index Lookup in Extended Tag Cache: readNext(Tag = t, Bound = B, Metalog position = p)

```
1: procedure On the compute tier
2:     if t not in Tag Cache then
3:         Return index miss
4:     end if
5:     ▷ Values of following tag fields (e.g. min_sqn) belong to t
6:     if B <= min_sqn then
7:         Set timestamp to p
8:         Return storage_shard_ids[0]
9:     end if
10:    if B < head of Suffix then
11:        if Complete == 1 then
12:            Set timestamp to p
13:            Return storage_shard_ids[1]
14:        else
15:            Return index miss
16:        end if
17:    end if
18:    if B > tail of Suffix then
19:        Return index cannot exist
20:    end if
21:    ▷ Binary-search in Suffix
22:    Get index i of sequence number s for which s is lowest upper bound of B
23:    Set timestamp to p
24:    Return storage_shard_id[i]
25: end procedure
```

shard in the storage tier.

For aggregating the results of index lookups from index nodes the index tier offers two modes. (1) In **master-slave-mode** one of the index nodes is the master. The other nodes are slaves and forward their best matches to the master. The master aggregates the matches. Selecting the master is done by the compute node on a per-request basis and in round-robin fashion to balance the load across index nodes. (2) In **aggregator-mode** the index tier contains apart from the index nodes one or more aggregator nodes which aggregate the best matches of the index nodes. Selecting the aggregator is done by the compute node on a per-request basis. If multiple aggregators exists the aggregator is chosen in round-robin fashion to balance the load across aggregators. Note that aggregators do not maintain any index data. *For simplicity, the rest of the design chapter only considers the aggregator-mode.*

5.3 Append Path

Figure 5.3 and Algorithm 5 illustrate the append path. Tasks in the compute tier issue $append(list\langle tags\rangle, record)$ calls and get back the unique sequence number assigned to that appended record. For brevity, the following explanation assumes a single tag per append instead of $list\langle tags\rangle$. The case with several tags can be derived by repeating the corresponding operations for each tag in the list.

All tasks on a compute node append to a single storage shard stored on one storage node and no other compute node writes to that shard. This requirement is necessary for the Scalog [Din+20] high throughput ordering protocol. In a nutshell, this allows compute nodes to keep local sequence numbers for their shard which can then be efficiently translated into global sequence numbers once the type-O metalog updates are received. This, however, need not lead to storage imbalances. First, a storage node can host many shards. Second, a compute node can start using a different shard, on a potentially different storage node, if the first shard is closed (cannot be appended to again).

Challenges

The index tier aims to balance the index data across the index nodes to avoid index node storage imbalances that can occur when some tags are often appended to. Another goal is to minimize the overhead of the type-S metalog updates. For instance, it is best avoided to send a metalog update to all index tier nodes.

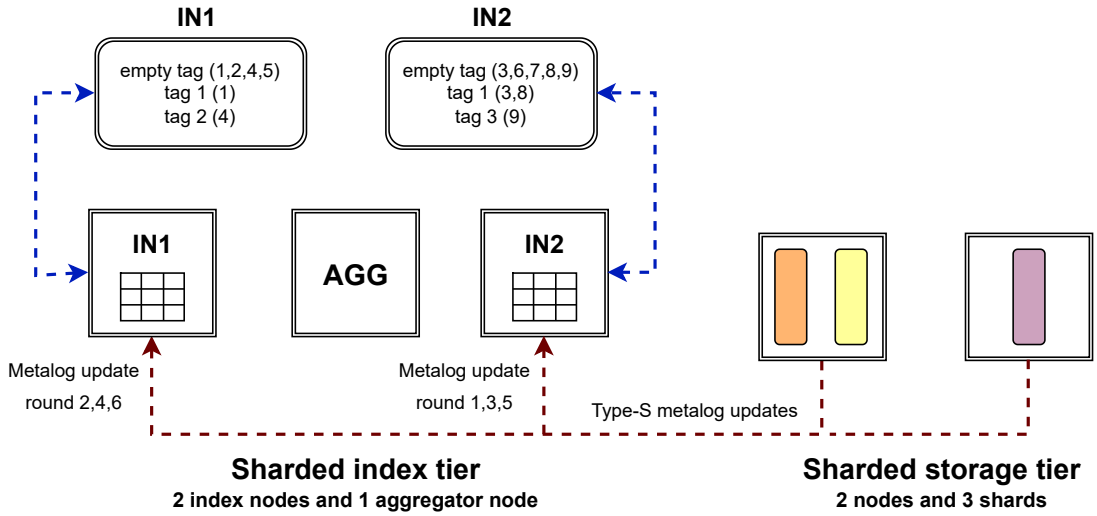


Figure 5.3: Integration of index-tier indexes on the append path in IndiLog . The top two rectangles illustrate the contents of the index in the index tier nodes.

5.3.1 Workflow

The index data is distributed over the index tier as follows. For every type-S metalog update, a single index shard is chosen in round-robin fashion to receive the metalog updates (Alg5:14-18). In Figure 5.3 there are two index nodes so one is chosen for the odd metalog update rounds and one for the even ones. Since metalog updates occur frequently (in Boki every 300 microsec) this ensures that index data is well distributed over the index tier. This also has several important implications which affect reads. Tags which are appended to over a longer period of time end up appearing over many metalog updates and thus can have their index data distributed over many (potentially all) index nodes (e.g., tag 1 in Figure 5.3). This avoids the undesirable situation when one popular tag uses a disproportionate amount of memory on a single index node. Tags which are appended to for a very brief period of time may have their index data on as little as one index node (e.g. tag 2 and tag 3 in Figure 5.3). Another implication is that a particular sequence number can end up being indexed on any index node (non-deterministic placement). However, a particular sequence number is indexed in only one index node.

Algorithm 5 also presents the steps taken on the compute tier. From the point of view of a function, an append blocks until a sequence number is returned. Sequence numbers are derived from type-O metalog updates (Alg5:5-6). The same metalog updates provide the necessary information to update the Suffix (Alg5:7). Eviction

Algorithm 5 Appends in IndiLog : SQN = append(Tag = t, record)

```
1: procedure On the compute tier(Append record)
2:     Send append to the corresponding storage shard
3: end procedure

4: procedure On the compute tier(On receiving type-O metalog update)
5:     Derive global SQNs
6:     Return SQNs to corresponding calling functions
7:     Incorporate metalog update into Suffix
8:     Evict index items in Suffix if threshold size reached
9: end procedure

10: procedure On the compute tier(On receiving type-S metalog update)
11:     Incorporate metalog update into Tag Cache
12:     Evict index items in Tag Cache if threshold sizes reached
13: end procedure

14: procedure On index tier node IN_IDX(On receiving type-S metalog update)
15:     if Metalog update round % nr_index_nodes == IN_IDX then
16:         Incorporate metalog update into local index
17:     end if
18: end procedure
```

occurs when the Suffix reaches a threshold size limit (Alg5:8). The Tag Cache is maintained by incorporating type-S metalog updates (Alg5:11). Eviction occurs when the Tag Cache reaches a threshold size limit (Alg5:12). Note that procedure Alg5:10-13 is not part of the append path but is implicitly triggered by append calls from functions as new index data are produced.

5.4 Read Path

Figure 5.4 and Algorithm 6 illustrate the read path. The tasks in the compute tier can issue two types of bounded reads for a tag t and a bound B: $readNext(t, B)$ and $readPrevious(t, B)$. The first type returns the record with the closest sequence number to b that is greater than or equal to B. The second type returns the record with the closest sequence number to B that is less than or equal to B. For clarity, Algorithm 6 only treats the first case. The second case is similar. Let OWL be the one-way network latency.

Algorithm 6 Reads in IndiLog : readNext(Tag = t, Bound = B)

```
1: procedure On the compute tier
2:     if t is Empty Tag then
3:         if B >= head of Suffix then
4:             Search in Suffix
5:         else
6:             Search in Popularity Cache
7:         end if
8:     else
9:         ▷ t is custom tag
10:        Search in Tag Cache
11:    end if
12:    if Local index lookup succeeded then
13:        Send read request to the identified storage shard
14:    else if Local index lookup failed then
15:        Send lookup to each index tier node tagged with the function's metalog update round
16:    end if
17: end procedure

18: procedure On the index tier - index nodes
19:    if Found B in the index then
20:        Forward read request for B to storage tier
21:        Send B to the aggregator
22:    else if B not found in index then
23:        Send to the aggregator closest BS, such that B < BS
24:    end if
25: end procedure

26: procedure On the index tier - aggregator nodes
27:    Receive BS from index node for read with unique ID RID
28:    if RID complete then
29:        Continue with next request
30:    end if
31:    if BS == B then
32:        Mark RID as complete. An index node already contacted the storage tier
33:        Continue with next request
34:    end if
35:    if all index nodes answered for RID then
36:        REC = min (all (BS))
37:        Forward read request for REC to index tier
38:    end if
39: end procedure
```

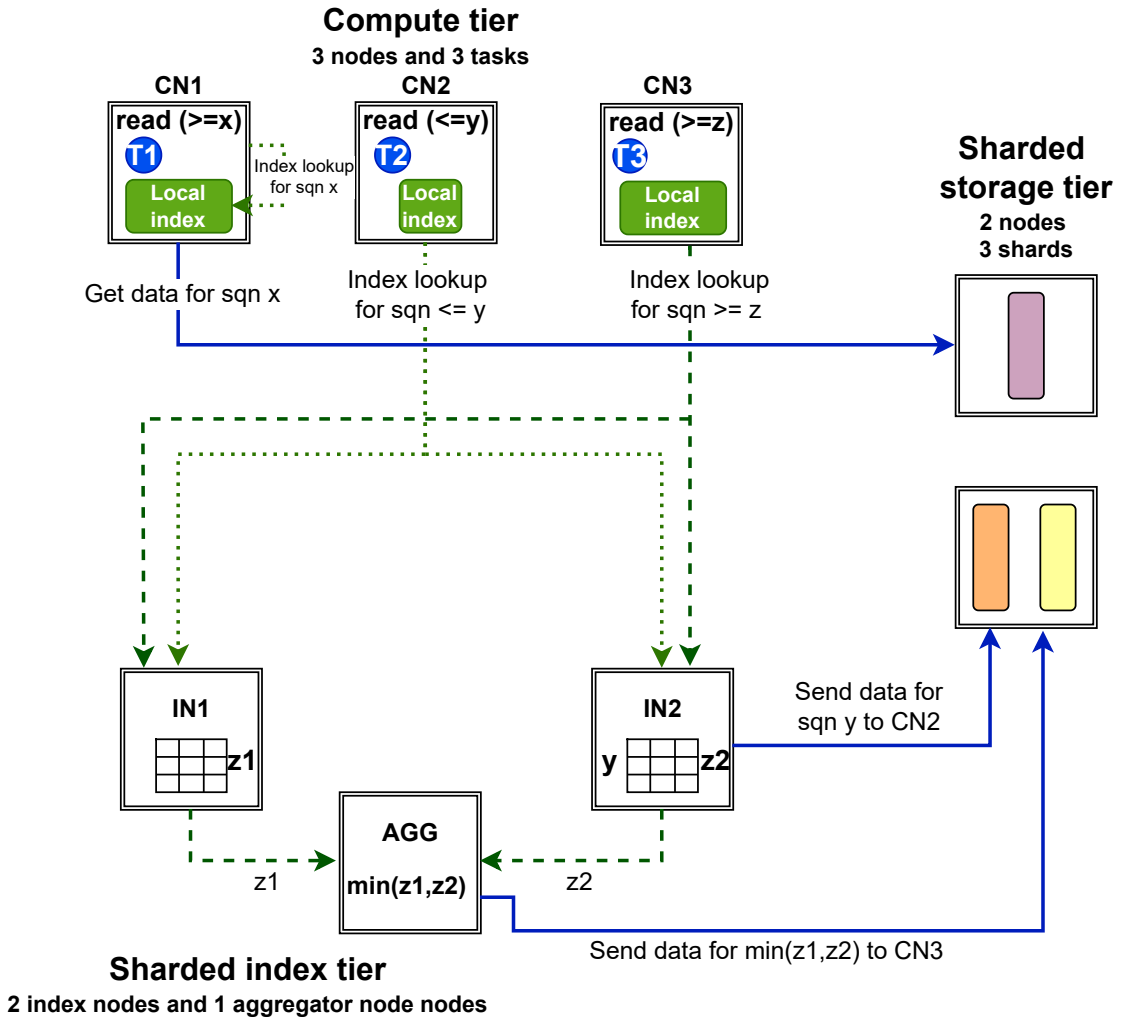


Figure 5.4: IndiLog Integration of indexes on the read path

Challenges

As described, the appends scatter the sequence numbers for a single tag, non-deterministically over the index tier in order to mitigate memory imbalances and reduce metalog update overheads. One particular sequence number can end up indexed on any index tier node. Moreover, custom tags may have a sparse list of sequence numbers i.e., some sequence numbers may be missing as they were assigned to other tags. When the bound is such a missing sequence number, bounded reads cannot return the bound but rather a closest match. Yet, index nodes know nothing about what other index nodes

store. All of these imply that the compute tier cannot direct a read towards a specific index tier node. Moreover, this implies that no single index tier node may be able to draw a final conclusion. This suggests the need to aggregate information from several index nodes.

5.4.1 Read Path Types

IndiLog has multiple read path types. Index lookups are either captured in the local index or must be forwarded to the index tier.

Read type 1 - Exact/closest match in local index

This corresponds to task T1 in Figure 5.4. In this case, the local index is able to locate the right sequence number. This can be an exact match when the index includes X (as in Figure 5.4) or a closest match when the index does not contain X but contains the sequence number that is the closest to X for that tag. Node CN1 then contacts the storage node hosting the record of the chosen sequence number (Alg6:12-13). This exchange takes $2 * OWL$ latency and 2 messages.

Read type 2 - Exact match in index tier

This corresponds to task T2 in Figure 5.4. After an unsuccessful local index lookup, node CN2 contacts each index node (Alg6:14-15). For consistency reasons, the request includes the function's metalog update round (discussed in §5.6). In this example $IN2$ happens to have an exact match (it indexed Y). $IN2$ immediately forwards the read request to the responsible storage node (Alg6:20) 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. Note that only a single index node can have an exact match due to the way the index data is distributed over the index nodes.

Read type 3 - Closest match in index tier

This corresponds to task T3 in Figure 5.4. After an unsuccessful local index lookup, node CN3 contacts each index node (Alg6:14-15). In this example, neither $IN1$ nor $IN2$ have an exact match. Neither stores Z. The index tier nodes search locally for the closest $Z\_local > Z$. They then send this to the aggregator (Alg6:23) which collects all results and finds the closest value to Z (the minimum of all aggregated Z_local) (Alg6:35-36). The aggregator then forwards the read request to the storage tier node

responsible for the identified sequence number (Alg6:37) which sends the data to T3. If the aggregator receives an exact match (Alg6:31) 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. This exchange takes $4 * OWL$ latency and the number of message is equal to $2 * (nr\_of\_index\_tier\_nodes) + 2$.

5.4.2 Interplay with Local Record Cache

Compute nodes can optionally store recently accessed records in a local record cache. The local index can have a subtle interplay with the local record cache. IndiLog assumes that a local index miss implies a local record cache miss. Because IndiLog currently does not attempt to keep the local record cache in sync with the local index, it is conceivable that the desired sequence number exists in the record cache but an incomplete local index is unable to definitely identify that sequence number as the desired one. Similarly, the index tier has no knowledge of local record caches so it only forwards requests to the storage tier.

5.5 Scalability of the Compute Tier

In IndiLog compute nodes can be be added or removed seamlessly. Any scaling operation of the compute tier does not initiate reconfiguration in any tier including the compute tier itself.

IndiLog leverages the storage shard identifier as the essential information that is exchanged between components. In IndiLog compute nodes hold storage shard identifiers (temporarily). An identifier allows the holder to append to the storage nodes associated with this identifier. A compute node may hold zero, one or multiple storage shard identifiers. However, an identifier can never be hold by two or more compute nodes simultaneously. This requirement is necessary for the Scalog [Din+20] high throughput ordering protocol. Furthermore, an identifier can be reused, if it is not hold by a compute node anymore. An identifier of a finalized shard that is read-only cannot be reused. The set of storage nodes that belong to the storage shard identifier can never change. The management of all identifiers is done by a central controller.

5.5.1 Registration Protocol for Compute Nodes

IndiLog introduces a registration protocol for compute nodes. A new compute node registers itself at nodes of the storage and ordering tier. Only after registration it can append records to the log and read records from it. The registration process includes

three steps. In the following we show the case for a compute node that requests a storage shard identifier which it can use after registration to append new log records.

In **step 1** the compute node asks the controller for a storage shard identifier. The controller takes one of the available identifiers *ss_id* and sends it to the compute node (in our design the controller takes the lowest identifier that is available). From now on this identifier is hold by the compute node and unavailable for other compute nodes until the compute node explicitly gives it free or terminates.

In **step 2** the compute nodes registers itself at all storage nodes. Storage nodes that belong to *ss_id* respond with a local sequence number. The local sequence number represents the number of records which the storage node persisted so far that belong to *ss_id*. All storage nodes must respond with the same local sequence number to avoid any inconsistencies. Therefore, a storage node drops log records belonging to *ss_id* which have not yet been acknowledged by a metalog update and thus are not persisted yet. The local sequence number is 0 if the storage nodes have not persisted any data yet for *ss_id*.

In **step 3** the compute node registers itself at the primary sequencer. The primary acknowledges the registration by sending the current metalog position to the compute node.

For a compute node that only reads log records step 1 is not necessary and the registration requests in step 2 do not contain any storage shard identifiers. However, steps 2 and 3 are mandatory: the storage nodes and the primary sequencer learn from the compute node's existence so that they can send metalog updates to the new node.

5.6 Other Design Properties

5.6.1 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 reads or appends. It is either (1) that of the last index it used for a read or (2) the last metalog update round that included a sequence number 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 that of the function. This may occur, e.g., if the local indexes get updated before the index tier.

However, this condition can be safely relaxed in IndiLog without jeopardizing the aforementioned guarantees. In IndiLog , the index nodes have different timestamps because of the round-robin nature of distributing metalog updates over the index nodes

(§5.3). That is why it is safe for an index node to participate in some index lookups for functions with a higher timestamp. In this case, the timestamp mismatch is not a sign of stale information. That index node is not meant to receive anything during the rounds when other index nodes are 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.

5.6.2 Failure Resilience

Since IndiLog borrows established distributed shared log concepts, it also borrows their failure resilience approaches. Replication is used for the ordering and for the storage tiers. As usual, the computing framework is assumed to be able to deal with compute node failures by restarting tasks. The loss of local indexes due to a compute node crash is not a concern because only that node was able to 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 elapses.

5.6.3 Scalability of the Index Tier

Given that index tier nodes never run functions, fewer index nodes are needed compared to compute nodes carrying complete local indexes. Therefore, we expect the scaling of the index tier to occur at a far coarser time granularity. Nevertheless, scaling the index tier is easy due to the round-robin approach in which metalog updates are distributed. A new index node can be easily added. All that is needed is for the storage and compute nodes to be informed about the new index node so that they can send it metalog updates and respectively send lookups to it. In IndiLog this is done via a configuration service like Zookeeper [Zoo].

Aggregator nodes can also be easily added since they only keep state for a short amount of time and on a per-request basis. All that is needed is for the index nodes to find out about the new aggregators. A special case occurs when an index node is running out of memory and thus needs to be removed from the round robin schedule for further metalog updates. This is done similarly via the configuration service. However, this node can and should still participate in index lookups and consistency is not an issue since it will not receive future updates.

5.7 Identifying New Tags

IndiLog has an optional solution to identify globally new tags i.e., tags that have never been used before across all compute nodes. If a tag can be classified as new, then any local lookup on this tag is a hit as long as no eviction for this tag happens.

5.7.1 Workflow

IndiLog extends the *default* append path (Algorithm 5) to identify new tags illustrated in Algorithm 7. If the append operation contains a tag that does not exists in the extended Tag Cache, then the compute node queries the index tier whether the tag is new (Alg7:2-5). The request goes to a single index node since the locations of minimum sequence numbers of custom tags are deterministic. The location of a minimum sequence number is determined by applying a modulo operation on the tag with the number of index nodes as divisor. The remainder points to the index node that holds the minimum sequence number (Alg7:3).

The compute node sends a bounded read to the index node: $readNext(t, 0)$ i.e., find the sequence number of tag t that is equal or greater and the closest to 0. Additionally, the compute node timestamps the request with the current tail of the shared log (Alg7:4). The index node receives the index lookup request. If its index does not contain the tag or it contains the tag but the minimum sequence number is higher than the timestamp, then the tag is new (Alg7:36-37), otherwise it responds that the tag is not new and sends the minimum sequence number of the tag (Alg7:38-39). The timestamp is necessary to counter a race condition in the append path: the metalog update that acknowledges the replication for the log record of the tag arrives at the index node before the index lookup for the minimum sequence number.

The compute node receives from the storage tier the index data of the replicated record that uses the tag (Alg7:21-34) and from the index node the minimum sequence number of the tag (Alg7:8-20). It combines the information such that either `min_sqn` of the tag is set to the head of `suffix` i.e., the tag is new and thus complete, or `min_sqn` is set to the minimum sequence number given in the index node's response.

Despite the advantage of increasing the local index hit ratio, the identification algorithm has important implications on IndiLog's index tier. First, it adds more load on the index tier. Compute nodes send queries to index nodes and type-S metalog updates are sent to multiple index nodes per metalog round because index nodes must store the minimum sequence numbers of new tags even if they are not selected for the metalog round (Alg7:43-47). Second, it impacts the scalability of the index tier. Since minimum sequence numbers of tags are placed deterministically on index nodes, changing the number of nodes leads to a remapping of all minimum sequence numbers

across the nodes of the index tier.

Algorithm 7 Appends in IndiLog with new tag detection: SQN = append(Tag = t, record)

```
1: procedure On the compute tier(Append record with custom tag T)
2:     if t not in Tag Cache then
3:         IN_IDX := t % nr_index_nodes
4:         Send readNext(t, 0) request and timestamp to index node IN_IDX
5:     end if
6:     Continue like in Algorithm 5
7: end procedure
8: procedure On the compute tier(On receiving min_sqn for t)
9:     if t is in Tag Cache then
10:        if t is new then
11:            Set min_sqn of t to head value of suffix of t
12:            Set complete_bit of t to 1
13:        else
14:            Set min_sqn of t to min_sqn
15:        end if
16:    end if
17:    if t is not in Tag Cache then
18:        Put response into list pending_min_sqns
19:    end if
20: end procedure
21: procedure On the compute tier(On receiving type-S metalog update)
22:    for custom tag t' in metalog update do
23:        if t' in pending_mins then
24:            Get min_sqn pending_min_sqns and remove entry
25:            if t' is new then
26:                Set min_sqn of t' to head value of suffix of t'
27:                Set complete_bit of t' to 1
28:            else
29:                Set min_sqn of t' to min_sqn
30:            end if
31:        end if
32:    end for
33:    Continue like in Algorithm 5
34: end procedure
35: procedure On index tier node(On receiving readNext(t, 0) request and timestamp s)
36:    if t not in index  or t in index  and s < min_sqn of t then
37:        Respond t is new
38:    else
39:        Respond t is not new and send min_sqn of t
40:    end if
41: end procedure
42: procedure On index tier node IN_IDX(On receiving type-S metalog update)
43:    for new tag t' in metalog update do
44:        if t' % nr_index_nodes == IN_IDX % nr_index_nodes then
45:            Store t' and min_sqn
46:        end if
47:    end for
48:    Continue like in Algorithm 5
49: end procedure
```

6 Implementation

IndiLog is built in C++ by adding the indexing architecture on top of Boki [JW21a]. Boki reuses the Scalog [Din+20] high-throughput ordering protocol. The functions running in the compute tier are written in Go and are using the approach from Nightcore [JW21b]. Compared with the release branch of Boki the implementation of IndiLog adds 7,892 new lines of code.

For our benchmarks we implement a library written in Go to run functions in IndiLog and Boki. Our experiments consist of Bash and Python scripts and files for declarative configurations. Our repository that contains the code is forked from Boki's benchmark repository as we reuse workloads of BokiFlow [JW21a] and BokiStore [JW21a]. Compared with the release branch of Boki's benchmark repository our repository adds 19,131 new lines of code.

Threading model

For running the functions generating appends and reads, IndiLog uses Goroutines. These are a form of green threads, a flow of execution managed and scheduled entirely by the Go language runtime running in user space. A number of Goroutines map to an OS thread. In our setup, every 32 Goroutines map to a single OS thread. Each compute node in IndiLog also has a number of IO threads running C++ code. These are OS threads that perform index lookups, read or write data and handle metalog updates. The Goroutines and IO threads communicate via Linux pipes (essentially FIFO queues) as in Nightcore. Therefore, several appends and reads may be queued waiting for an IO thread.

Some of the function calls are split between several phases. For example, for reads type2/3 (§5.3), the first phase involves checking the local index and contacting the index tier. At this point, the OS thread is free to serve other requests. The second phase of such reads is executed at a later time when data is actually received from the storage tier. All appends are also split. The first phase involves sending the data to the storage tier. After this, the OS thread becomes free. The second phase consists in receiving the sequence number via a metalog update from the primary sequencer.

Locking

Locking the index data structure is needed so that metalog updates do not impact reads. The index data structure 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 when reads or metalog updates occur. Since in our evaluation we compare against Boki, for fair comparison, in IndiLog we reuse the same coarse-grained locking for both the local indexes and the index tier. The Popularity Cache is an exception to this as it is not maintained by incorporating metalog updates.

For combining index results in the master-slave-mode and in the aggregator-mode we use fine-grained, per-request locks. When IndiLog activates the identification of new tags (§5.7), index nodes use an additional data structure that maps tags to minimum sequence numbers and manages locks on the level of map entries. Data structures that let us implement fine-grained locking are taken from the oneTBB library [Int].

Local Index

For the implementation of the Suffix and the Tag Cache we use data structures from the C++ Standard Library and from the Abseil libraries [Goo]. We use for the Popularity Cache a LRU cache taken from the Tkrzw library [Hir]. It is sharded internally to improve concurrent access.

Storing

IndiLog reuses Boki's implementation of storage nodes. New records are first kept in RAM and get eventually stored on disk by using RocksDB [Met]. IndiLog reuses Boki's optional record cache in compute nodes. It is a LRU cache taken from the Tkrzw library [Hir].

Message ordering

Applying metalog updates in the correct order is crucial for data structures that incorporate metalog updates. However, messages may take different network paths and it may happen that a higher (younger) metalog update arrives before a lower (older) one. IndiLog reuses mechanisms from Boki to preserve delivery order. Each data structure in IndiLog that receives metalog updates knows which metalog update comes next and puts out of order messages in a pending list.

Configuration management

IndiLog uses ZooKeeper [Zoo] to store global configurations and to manage the dynamic scaling of the compute tier.

7 Evaluation

We evaluate IndiLog in a benchmark study to find answers to the following research questions:

- What effects do we observe when we scale the compute tier of IndiLog?
- How is IndiLog impacted by different workloads and different levels of contention?
- How does the index tier of IndiLog performs and how does it scale?
- How does IndiLog behave when it runs for a long period of time?
- How does IndiLog perform when it identifies new tags on the append path?
- How does IndiLog behaves for workloads generated by real applications built on top of IndiLog?

In our studies we compare IndiLog with Boki [JW21a], a state-of-the-art distributed shared log.

7.1 Methodology

Setup

We use cloud VMs with 16GB of RAM and 4vCPUs running Ubuntu 20.04 with Kernel version 5.10.0. The mean latency between VMs as measured by ping is $180\,\mu s \pm 40\,\mu s$. The bandwidth between two VMs as measured by iperf is 2,127 Mbps.

For the storage tier the number of nodes is equal to the number of compute nodes and the replication factor is 1, e.g., if we use 4 compute nodes then we use 4 storage nodes. 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 microseconds. 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 IO threads. Index nodes, storage nodes, sequencer nodes and the aggregator node have 2 IO threads. The Suffix stores up to 10^5 entries. The Popularity Cache can hold up to

10^4 entries. The Tag Cache stores up to 10^6 sequence numbers over all tags. A single tag in the Tag Cache is limited to 10^4 sequence numbers. This local index configuration in IndiLog limits the size of a local index to ∼20 MB. The indices are stored in RAM on both compute nodes and the index tier. By default, local record caches are not used because their behavior is predictable and we are interested more in the behavior of requests that require communication between tiers.

Metrics

We present median and 99th percentile tail latency for reads and appends. The latency measurement starts when an IO thread first picks up a specific request and ends when the IO thread finishes the work for the request and the request can now return to the Goroutine that started it. Therefore, the latency measurement does not include any queueing time at the user-level waiting for service. The latency does include queueing time inside the system in the case of requests that comprise of two phases (§6). We chose to focus on this latency (as opposed to user-perceived latency) because our goal is to understand the scalability and the behavior of the system under challenging workloads and at high throughput.

We also show the global throughput, across all the compute nodes in the system. We also present various statistics (e.g., index hit rates) and breakdowns (e.g., different types of reads).

Workload

Our workloads are a mix of reads and appends. We use either a balanced workload (50% reads, 50% appends) or a read-heavy workload (95% reads, 5% appends). The workloads have either high concurrency (15 OS threads/compute node, 32 Goroutines/OS thread) or low concurrency (4 OS threads/node, 32 Goroutines/OS thread) to run the functions which call reads and appends.

Our functions are intentionally trivial (they just generate appends and reads) because we are interested in challenging the system. Generally, the appends stress the system via metalog updates which require locking and network communication. Reads stress the system via index lookups and also require locking.

We distinguish between the empty tag and a custom tag. All records have the empty tag, therefore the empty tag corresponds to the list of all sequence numbers in the log. A custom tag forms a sub-stream in the log and corresponds to a list of sequence numbers that are often non-consecutive. Records sizes are 1KB. There are several ways to choose a sequence number to read from a tag. Our workloads use a mix of accesses (e.g. the first sequence number, 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.

Workloads for the experiments 7.6.1 and 7.6.2 are application-specific and we describe them in their corresponding section.

Competitor

We compare IndiLog against Boki [JW21a], the state-of-the-art distributed shared log built specifically for serverless workloads. Boki uses complete indexes local to the compute nodes running functions. As shown (§3), the complete indexes can easily cause OOM errors. In the following experiments we purposely avoid that by running the experiments for a shorter period of time. Unfortunately, Boki cannot dynamically scale the compute tier. Nevertheless, for a fair comparison, we start Boki in the configuration that it would end up if it was able to scale. As in §3, scaling in a Boki-like design can be done by allowing the new nodes to access remotely the indexes on existing nodes. As in §3, for Boki we distinguish between Boki-hybrid nodes (running workload and hosting an index), Boki-no-index nodes (running a workload but not hosting an index and therefore needing remote index lookups) and Boki-index-only nodes (hosting an index but running no workload).

7.2 Scaling the Compute Tier

In all experiments we scale IndiLog and Boki from 1 to 4 compute nodes except from experiment 7.2.4. In Boki, the 3 new nodes query the index hosted one the first node. Because Boki cannot dynamically scale, it is already started in the 4-node configuration where it would end up if it was able to dynamically scale. In IndiLog, the 3 new nodes, each hosting a local index, join at second 30.

7.2.1 High Hit Ratio in Local Indexes

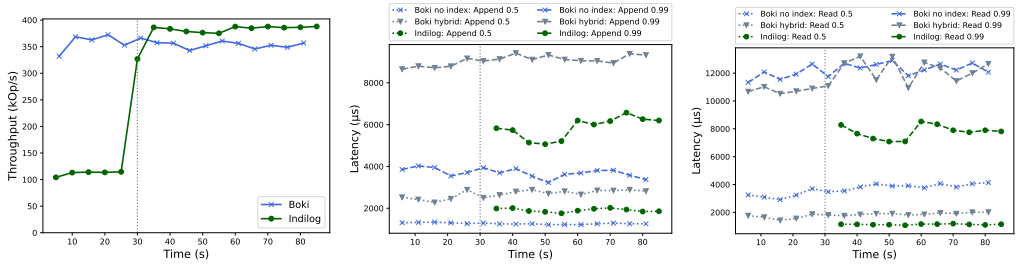
The experiment in Figure 7.1 shows the impact of scaling the compute tier from 1 to 4 compute nodes. We use a workload in which the local indexes on IndiLog nodes catch most of the lookups but a small portion still go to the index tier. In this experiment, the hit ratio in the local indexes for IndiLog across all 4 nodes is 87%. The local hit ratio for Boki is 100% because it uses complete local indexes.

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 previously appended tag but may or may not be related to the sequence number of the last append. For reads, we use a mix of accessing a tag. For an empty tag we read with equal probability (1) the appended value, (2) a popular sequence number, (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 7.1a shows the throughput across all 4 nodes averaged every 5 seconds. IndiLog is able to dynamically scale well from about 110 KOp/s to 390KOp/s. The 4 IndiLog nodes have comparable throughput. In contrast, Boki can only obtain 350KOp/s. Its nodes have a skewed throughput. The Boki-hybrid node achieves only 60KOp/s on average due to the contention. The other Boki nodes obtain about 96KOp/s. Figure 7.1b shows median and tail (p99) latencies for appends while Figure 7.1c show median and tail (p99) latencies for reads. As expected, the tail latencies 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 these nodes do not have a local index and thus do not pay the overhead of managing it. For reads (Figure 7.1c), the reads on the Boki-hybrid node are faster than for the Boki-no-index because the latter requires a remote index lookup. Finally, the reads in IndiLog show the best latency because IndiLog is not impacted by a contention similar to the one on the Boki-hybrid node.



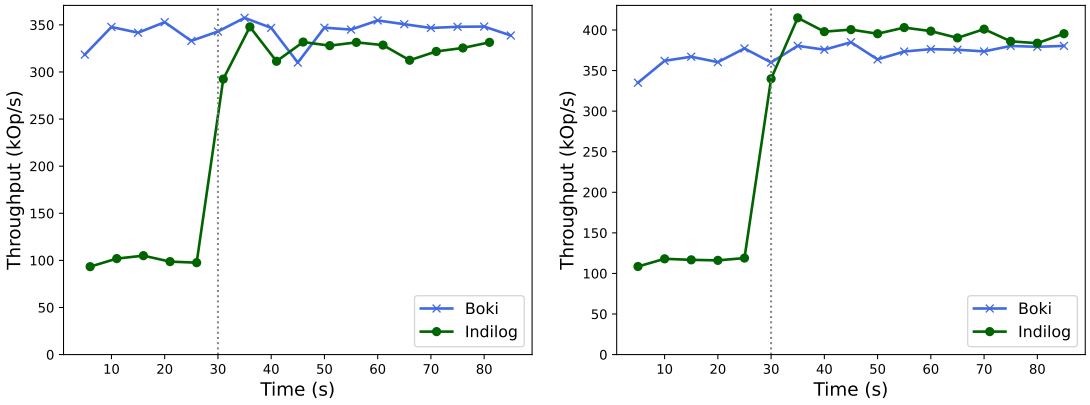
(a) Throughput (b) Append latencies (c) Read latencies

Figure 7.1: Throughput and request latencies (y-axis) when scaling the compute tier (at sec. 30 on the x-axis) from 1 to 4 nodes. High local index hit ratios in IndiLog. Boki cannot scale dynamically so for Boki we start by using 4 nodes.

7.2.2 Low Hit Ratio in Local Indexes

This experiment shows a challenging case for IndiLog when the vast majority of local index lookups fail and need to be serviced by the index tier. Still, IndiLog behaves competitively or better than Boki. Figure 7.2 shows the throughput for two workloads: a balanced read-append workload and a read-heavy workload.

The local index hit ratio in IndiLog is 20%. In Boki, the hit ratio is still 100% because of the complete indexes. To obtain low hit ratios we use a workload with only custom new tags. Reads are either from the head of the log (with 80% probability) or of the last appended value (with 20% probability). Note that this results in 80% of reads of type 3 (as described in §5.4) which are the most expensive reads in IndiLog because they need the aggregator node to decide on the closest sequence number.



(a) 50/50 Read/Append

(b) 95/5 Read/Append

Figure 7.2: Throughput when scaling the compute tier (at sec. 30 on the x-axis) from 1 to 4 nodes. Low local index hit ratios in IndiLog.

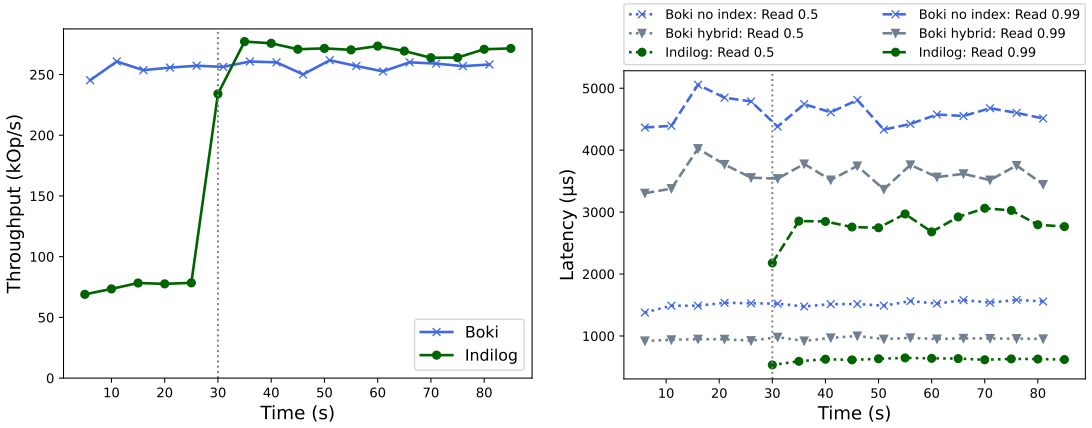
Figure 7.2a shows lower throughputs than Figure 7.2b for both Boki and IndiLog because it has a larger ratio of appends and they tend to be the more expensive operation. Despite the low index hit ratio, IndiLog shows higher throughput than Boki in Figure 7.2b. In Figure 7.2a, Boki has a slight edge because only the Boki-hybrid node pays the overhead of maintaining the local index via metalog updates. In contrast, in IndiLog, all 4 nodes pay this overhead.

7.2.3 Impact of Lower Concurrency

This experiment builds on §7.2.1 by changing the workload concurrency. Instead of each compute node running 15 OS threads each with 32 Goroutine, this experiment has

each compute node run only 4 such OS threads. This setup benefits Boki by lowering the contention on the Boki-hybrid node. The change of the workload can be best seen by comparing the throughput in Figure 7.3a and Figure 7.1a. The one in Figure 7.3a is significantly lower due to the lower workload concurrency.

IndiLog still comes out on top. Boki is helped by the less intensive workload but this also reduces the overheads that IndiLog nodes have to maintain their local index. Figure 7.3b shows read latencies. Compared to Figure 7.1c, the less intensive workload results in much lower latencies. We omit the append latencies but they show a similar pattern.



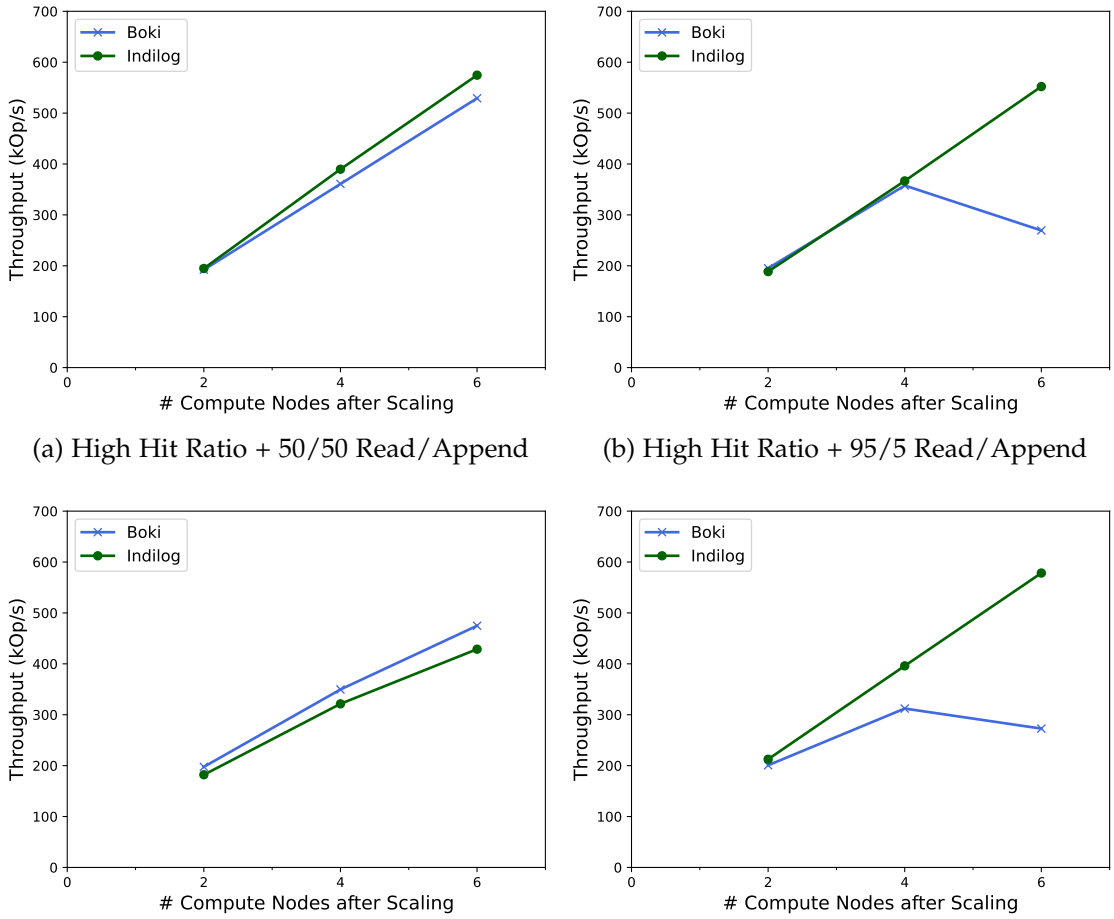
(a) Throughput

(b) Read latencies

Figure 7.3: Throughput and read latencies for lower workload concurrency (4 * 32 Goroutines per node).

7.2.4 Impact of Varying the Scaling Size

Figure 7.4 analyzes the impact of scaling the compute tier to a smaller or larger number of nodes. The previous experiments showed the case with $X = 4$. We use a balanced and a read-heavy workload, both configured with a high (87%) and a low (20%) local index hit ratio for IndiLog , respectively. IndiLog scales well, almost linearly for all workloads. In contrast, Boki scales well only for a balanced workload. For a read-heavy workload behaves increasingly worse as the size of the scaling increases $(X = 6)$ due to increased pressure on the hybrid node for remote index lookups. In contrast, when scaling with fewer nodes $(X = 2)$ the two systems behave similarly.



(a) High Hit Ratio + 50/50 Read/Append

(b) High Hit Ratio + 95/5 Read/Append

(c) Low Hit Ratio + 50/50 Read/Append

(d) Low Hit Ratio + 95/5 Read/Append

Figure 7.4: Varying the scaling size for different workloads. The number of compute nodes grows from 1 to 2/4/6 by adding 1/3/5 new nodes.

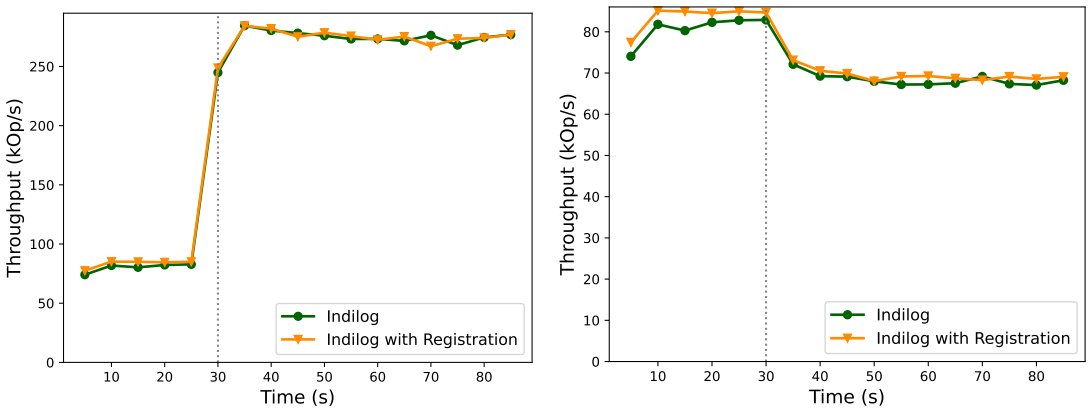
7.2.5 Impact of the Registration Protocol

In the previous experiments new compute nodes in IndiLog had been already registered when they joined at second 30. This experiment evaluates the performance impact when new computes nodes must register first before they can operate (see §5.5.1). Our results show that the registration process has no implications on the performance of IndiLog.

We use again a workload with a high local index hit ratio (87%) but with a low concurrency (4 * 32) to reduce contention in the system and single out the load from

the registration process. We have two different configurations of IndiLog: in the first one the compute nodes are already registered (default configuration). In the second one the compute nodes must register at the nodes of the other tiers when they join at second 30.

Figure 7.5a shows the overall throughput. The registration effort has no significant impact on the throughput. Both configurations show an almost identical throughput when scaling happens. Figure 7.5b takes a glance on the throughput of the old node i.e., the node that operates already at the beginning of the experiment. The throughput drop by ∼10-15 kOp/s of the old node when IndiLog scales is expected in both configurations. Both lines in Figure 7.5b reflect the circumstance that the ordering, storage and index tier receive more load when compute nodes are added. Despite that, we cannot see any visible performance implications when new nodes register. The throughput of IndiLog with registration is even slightly better compared to the default configuration because of variations in the cloud network of our VMs.



(a) Total Throughput

(b) Throughput of Old Node

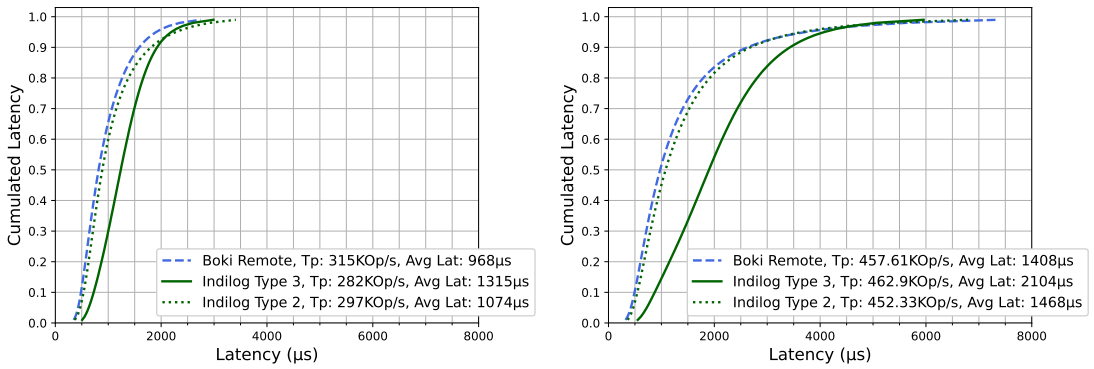
Figure 7.5: Throughput when scaling the compute tier (at sec. 30 on the x-axis) from 1 to 4 nodes. The throughput of the old node decreases in both configurations because the system gets under higher load. The registration process of the new nodes has no visible impact.

7.3 Design of the Index Tier

The following benchmarks focus on IndiLog's index tier.

7.3.1 Breakdown of Read Latencies

This experiment singles out the performance of the different types of reads in IndiLog. As shown in §5.4, IndiLog has 3 types of reads but of particular interest are the reads that are not present in Boki i.e., the reads of type 2 and 3 which go to the index tier. To single them out we use a read-heavy workload with 5% appends and 95% reads and we remove the local indexes from the IndiLog nodes. Thus, in this experiment, all reads in IndiLog need the index tier. For comparison, since IndiLog uses 2 index nodes, we give Boki 2 Boki-index-only nodes with complete indexes which also do not run workloads. 4 other Boki-no-index nodes send requests to them.



(a) Low concurrency

(b) High concurrency

Figure 7.6: Latency CDFs for IndiLog reads using index-tier lookups.

Figure 7.6 shows the latency CDFs for both the high and the low concurrency workload. 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 both the tail latencies in all cases and also the gap between the type 2 and type 3 reads.

7.3.2 Scalability of the Index Tier

In this experiment we show how IndiLog behaves when we scale the index tier. Therefore, we increase the number of index nodes as well the number of aggregators. Table 7.1 shows the configurations and the resulting throughput. Scaling the index tier is mostly relevant for type 3 reads where the aggregator(s) need to collect best matches from all other index tier nodes before forwarding the result to the storage tier. Therefore, we reuse the high concurrency workload in §7.3.1.

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 the aggregator is not the bottleneck. The slight differences between 1 and 2 aggregators are explained by natural variations caused by variations in the cloud network latency.

Index Tier Configuration	Throughput [kOp/s]
S:2, R:1, A:1	465.2
S:2, R:1, A:2	461.4
S:6, R:1, A:1	385.4
S:6, R:1, A:2	389.5

Table 7.1: Throughput evaluation of scaling the index tier in IndiLog . S is the number of index shards. R is the replication factor. A is the number of aggregators.

7.3.3 Benefit of Using an Aggregator

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 selected randomly as the master in the index lookup request from the compute tier. With a dedicated aggregator node, for the high concurrency workload, the throughput is 465.2KOp/s (as in Table 7.1) and the median latency is 1.76ms. Without a dedicated aggregator node, the throughput is 440.6KOp/s and the median latency 2.07ms. The trend is similar for the low concurrency workload: 282.4 KOp/s and 1.23ms with the aggregator node and 237.9KOp/s and 1.62ms without the aggregator. We expect the gap between the two solution to increase for more index nodes. The results are expected as the aggregator node reduces the burden on the index tier nodes.

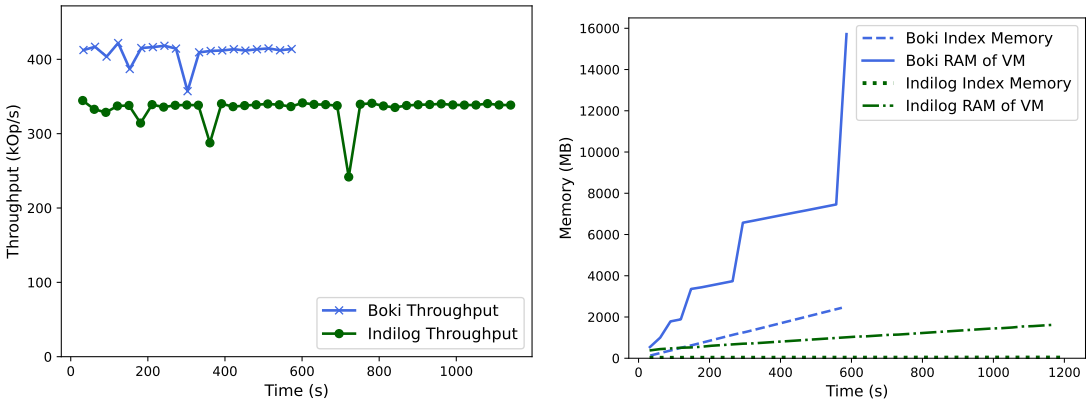
7.4 Long-term Running

We run a index data intensive benchmark over a longer period of time. IndiLog and Boki use 4 compute nodes that run functions, respectively. Boki's compute nodes maintain complete indexes. The workload creates new tags continuously: each append call uses a new custom tag and is followed by one read for this tag. We access the tag either from the head of the log or use the sequence number from the append call such that IndiLog has a local index hit ratio of 80%. Thus, 20% of reads in IndiLog are of type 3 and need to be serviced by the index tier.

We showed in §3.2.1 that Boki's complete indexes quickly exhaust the VM memory leading to OOM crashes. As expected, we see this again in Figure 7.7: Boki fails

after ∼10 minutes due to OOM crashes in the compute nodes. The compute nodes in IndiLog evict index data to keep the memory footprint on a low level. IndiLog still runs after 20 minutes. Note that the RAM in a IndiLog's compute node slightly grows over time as it continuously collect statistics for benchmarking.

In Figure 7.7a we observe temporary drops in throughput for Boki and IndiLog, respectively. Complete indexes in Boki's compute nodes and sharded indexes in IndiLog's index nodes reallocate memory for their index data structures. During a reallocation append and read operations are blocked from accessing the data structures. As RAM grows the reallocation takes more time and drops get bigger. This can be best seen by comparing the relative times of Boki's drops in Figure 7.7a and the big jumps in RAM usage of Boki's compute node in Figure 7.7b.



(a) Throughput over time

(b) Memory usage over time on a single compute node

Figure 7.7: Throughput over time in IndiLog and Boki. IndiLog's local index hit ratio is 80%. Compute nodes in Boki have complete indexes (100% local hit ratio) and crash after 10 minutes because of OOM.

7.5 Impact of Identifying New Tags

We evaluate how IndiLog performs when its compute nodes identify new tags as described in §5.7. We use two configurations of IndiLog : IndiLog-Extended with new tag identification and IndiLog-Default without. Our results show that despite a higher local index hit ratio for IndiLog-Extended the additional effort to identify new tags outweighs the advantage of a higher hit ratio.

We use two workloads: (1) a mixed workload which we used already in the previous experiments for which both configurations achieve high local index ratios and (2) a

heavily skewed workload. In (2) each append contains a new unique tag and is followed by a read of the tag from the head of the log. Under these conditions IndiLog-Default has not any local index hits because the Tag Cache cannot guarantee for a tag that the first sequence number in the tag's `Suffix` is the minimum. On the other hand, IndiLog-Extended theoretically should have always hits because it determines the minimum sequence number of each tag. We run the experiment multiple times and increase the workload concurrency each time. We start with a very low concurrency (1 * 32 Goroutines per node) and finish with a very high concurrency (15 * 32 Goroutines per node).

The upper row of Figure 7.8 shows the results of workload (1). IndiLog-Default performs better than IndiLog-Extended for all concurrency levels. The local index hit ratio of IndiLog-Default and IndiLog-Extended are 86% and 97%, respectively. As expected, IndiLog-Extended is able to capture more index requests locally.

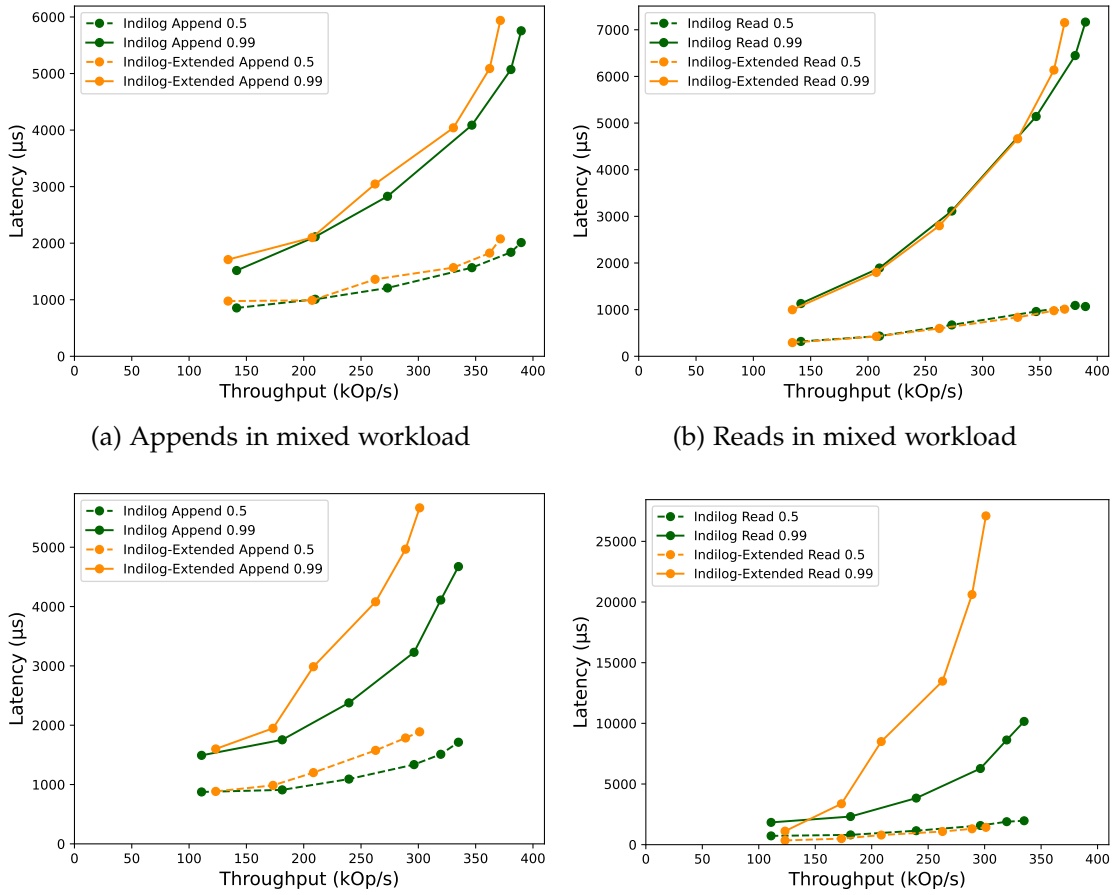
The lower row of Figure 7.8 shows the results of workload (2). For a very low concurrency IndiLog-Extended performs better. However, we see that IndiLog-Extended gets significantly worse with higher concurrency. As the workload creates always new tags the compute nodes must always send queries to the index tier to find out the minimum sequence numbers of tags, though the index tier does not store them yet. This adds high load on the index tier. Moreover, compute nodes must incorporate results for minimum sequence number into the extended Tag Cache which uses coarse grained locking. Although IndiLog-Extended captures almost all the index lookups locally, this does not outweigh the effort to determine new tags. Moreover, the local index hit ratio decreases with higher concurrency: from 100% for the lowest concurrency level to 89% for the highest concurrency level. The 11% miss ratio reflects the high load on the index tier: although the queries to determine minimum sequence numbers need only a single round trip time, in 11% of all cases the queries are so slow that the append path of the new tags and the followed reads for these tags are faster than receiving and incorporating the responses of the queries.

7.6 Realistic Workloads

We present how IndiLog behaves for real applications.

7.6.1 Object Storage Workload

In this experiment we show how the limited local indexes in IndiLog and the continuous eviction from them affect performance. The main message is that IndiLog can obtain performance comparable to Boki in a real application even with local indexes that are far smaller than Boki's complete local indexes.



(a) Appends in mixed workload

(b) Reads in mixed workload

(c) Appends in heavily-skewed workload

(d) Reads in heavily-skewed workload

Figure 7.8: Throughput vs latency comparison between IndiLog and IndiLog-Extended. IndiLog-Extended identifies new tags. The upper row shows the results generated from a mixed workload for which both configurations achieve a high local index hit ratio. Results of the lower row are generated from a heavily skewed workload in which new tags are always created and reads lookup only the minimum sequence numbers of tags.

We run IndiLog and Boki as the infrastructure layer of an object storage library. We reuse the BokiStore [JW21a] library that stores objects durably on shared logs and provides transaction support. As explained in §2.3.3 BokiStore uses customs tags for all log operations. As the workload for the object storage library we reuse Boki's Twitter clone [JW21a]. The workload runs functions to login users, publish tweets, show timelines and see user profiles. For our experiment we initialize the workload

with 10,000 users and use then 192 concurrent clients to trigger these functions via HTTP requests. For this experiment we enable the local record cache in IndiLog and Boki. The measurement of throughput and latencies is done at the clients.

Figure 7.9 shows how the size of the local index affects the hit ratio. In the left bar chart we configure IndiLog with the default limits from §7.1. In the right bar chart we configure IndiLog with a smaller index (we call this IndiLog-Small): we set the sequence numbers limit of the tag cache to 10^4 and a single tag can only have 100 sequence numbers until eviction kicks in.

With the default limits IndiLog successfully handles 93% of all index lookups in the local index. Since there are many evictions in IndiLog-Small we observe that the local index hit ratio decreases. However, even with its small index (only 0.2MB) almost 50% of all index lookups are handled locally.

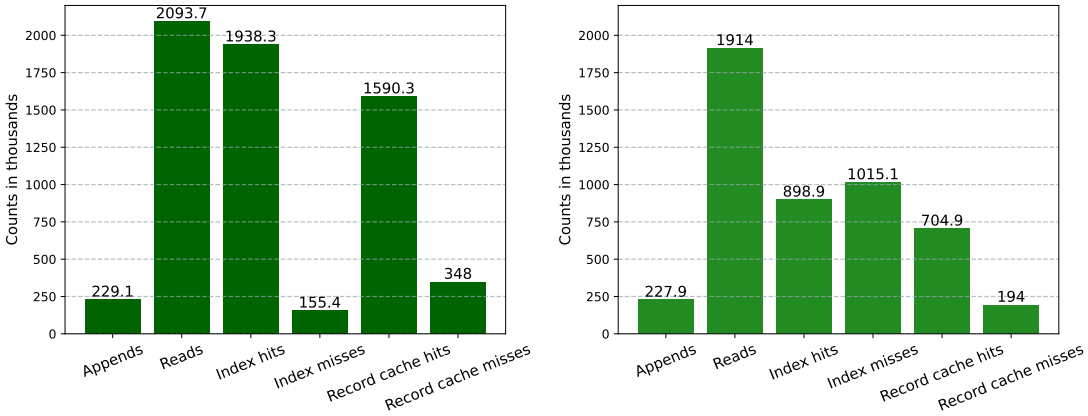
We next compare IndiLog with Boki. Each system uses 4 compute nodes to run the functions. In IndiLog we use again the two configurations for the local index: default and small. For Boki we also use two configurations: complete and remote. In the former configuration all compute nodes in Boki maintain a complete index. In the latter, the 4 compute nodes that run functions have no index and must do remote index lookups to 2 other compute nodes that do not run functions but have the complete index. Tables 7.2 and 7.3 compare both systems in terms of throughput and latencies. As expected, Boki-Complete performs the best because the hit ratio of Boki-Complete trivially is 100%. Its record cache can serve 76% of all reads which is similar to IndiLog with default index limits. Due to evictions, IndiLog-Small has lower throughput and higher latencies compared to IndiLog. However, IndiLog-Small performs better in overall throughput compared to Boki-Remote.

System	Throughput [Op/s]
IndiLog	8700.1
IndiLog-Small	8430.5
Boki-Complete	8950.9
Boki-Remote	8381.4

Table 7.2: Throughput in the object storage workload.

7.6.2 Fault-tolerant Workload

In this experiment we show how IndiLog performs for a real application for which IndiLog has a low local index hit ratio. Moreover, we see the effect of an extended Tag Cache (IndiLog-Extended) that is able to capture more index lookups. We finally



(a) Default index size (∼20 MB)

(b) Small index size (∼0.2 MB)

Figure 7.9: Aggregated statistics over all compute nodes in IndiLog with different index size configuration at the compute node for the object storage library workload.

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

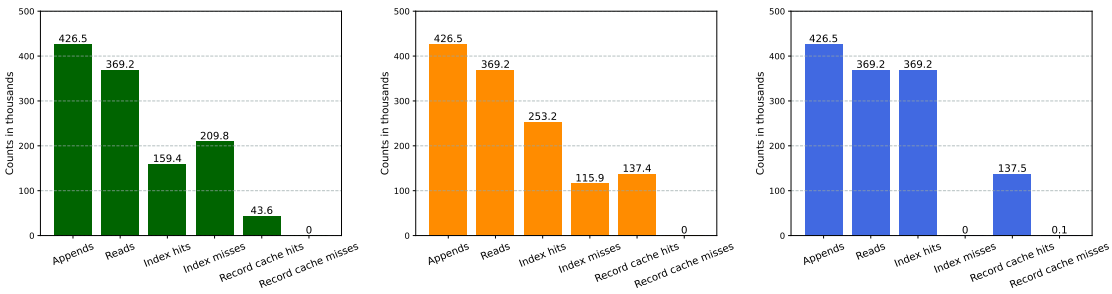
Table 7.3: Latencies (in msec.) in the object storage workload.

summarize that despite a low index hit ratio in IndiLog the performance is comparable to IndiLog-Extended and Boki's complete indexes.

We run IndiLog and Boki as the infrastructure layer for a faul-tolerant workflow library, both equipped with 4 compute nodes. Moreover, we use two configurations for IndiLog: default and IndiLog-Extended. The latter identifies new tags and uses the extended Tag Cache. In Boki all compute nodes maintain a complete index. As

application we reuse the BokiFlow [JW21a] library that guarantees exactly-once semantics and allows for idempotent database updates. As explained in §2.3.3 the library queries the minimum sequence numbers of tags to distinguish between remaining and completed steps of a workflow. This disfavors IndiLog (default) as its Tag Cache can never guarantee that the first sequence number in the `suffix` of a tag is indeed the minimum sequence number. For the experiment we use Boki's workflow workload for travel reservations. Boki uses the workload from Beldi [Zha+] which adapts it from DeathStarBench [Del] microservices: multiple functions let clients book flights and hotels, rate them, etc.. However, the workload requires access to DynamoDB [Amaa] on AWS's cloud platform. In our experiment we setup DynamoDB locally on one of our VMs. The local DynaomDB impacts the performance, however, this is acceptable as it affects IndiLog and Boki equally. The workload keeps 2 HTTP connections open and runs a constant benchmark with 150 requests per second to trigger the functions. Latencies are measured at the gateway.

Bar chart 7.10 compares Boki, IndiLog and IndiLog-Extended by looking at the aggregated statistics over their compute nodes. Boki's local index hit ratio is trivially 100%. As expected, we observe that IndiLog-Extended is able to capture more index lookups (69%) than IndiLog (43%). IndiLog-Extended has with 37% an identical record cache hit ratio compared to Boki. Latencies in Table 7.4 reflect the statistics of bar chart 7.10. Boki performs best due to the complete indexes whereas IndiLog has the highest latencies because it must consult the index tier often. Nevertheless, IndiLog achieves a comparable performance.



(a) IndiLog (b) IndiLog-Extended (c) Boki

Figure 7.10: Aggregated statistics over all compute nodes for the workload of the fault-tolerant workflow library. Left: IndiLog with default configuration. Middle: IndiLog with activated new tags identification. Right: Boki with complete indexes.

Table 7.4: Latencies (in msec.) in the fault-tolerant workflow workload.

	IndiLog		IndiLog-Extended		Boki	
Latency	50th	99th	50th	99th	50th	99th
msec	104.7	195.3	101.3	189.6	98.7	185.1

7.7 Summary of the Findings

In all scaling experiments we find that the compute tier of IndiLog scales linearly in almost all cases. It shows 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, (3) varying the scaling size (i.e., the number of new compute nodes added to the system) and (4) using read-heavy workloads.

Our experiments show that IndiLog's sharded index tier shows comparable performance to complete indexes in Boki. Moreover, even for a high number of index shards the index tier has an acceptable performance.

IndiLog keeps memory consumption of local indexes on a low level. Therefore, when running an experiment over a longer period of time the compute nodes in IndiLog do not experience OOM crashes like those in Boki.

We see that IndiLog's optional new tag identification has no significant advantage: a higher local index hit ratio cannot outweigh the costs that it takes to identify new tags in the first place. Especially in workloads with many new tags and high concurrency levels the load on the index tier is too high.

Finally, we find that IndiLog performs well for realistic workloads with an (incomplete) local index. Even with a very small index IndiLog can capture many index lookups. For workloads that disfavor IndiLog's incomplete local indexes IndiLog still has comparable results to Boki with complete local indexes.

8 Related Work

8.1 Serverless State Management

There is significant related work on using other storage abstractions, apart from distributed shared logs, for serverless state management. Neither of these works focuses on an indexing architecture for efficiently accessing the storage.

The first approach to sharing state between functions was to leverage cloud object stores such as Amazon S3. Unfortunately, this was shown to be very inefficient [PVS]. As a result, Locus [PVS] proposes to benefit shuffles between functions by leveraging a mix of slow but cheap storage medium (e.g. Amazon S3) coupled with a small amount of memory-based fast storage to bring benefits in performance while remaining cost effective.

Beyond improving state management of functions for cloud object stores various systems were introduced motivated by the needs of stateful functions. The following selection of systems depicts only a small fraction of recent development.

Cloudburst

Cloudburst [Sre+20] uses Anna [Wu+], an autoscaling key-value store for state sharing combined with caches co-located with the functions. Anna uses a distributed index co-located with the storage that maps keys to the caches that store it. Anna can then use this index to propagate key updates to caches. IndiLog's index is different since it indexes the storage and not the function-level caches.

Pocket

Pocket [Kli+] is a distributed data store targeted at the ephemeral data used by serverless functions to share state. Pocket uses multiple tiers (e.g., DRAM, flash, disk) and provides an elastic and cost-effective storage solution.

Faa$t

Faa$t [Rom+] is a distributed in-memory cache integrated into the FaaS runtime to reduce remote storage reads of functions. When a function is loaded it gets equipped

with a Faa$t cache. The cache is transparent for the function and may be pre-warmed with data that is likely to be accessed by the function. An ownership mechanism guarantees consistent reads between multiple cache instances that share state between functions: a local miss for an object that is owned by the local cache results in a remote storage read. A local miss for an object that is not owned by the cache is delegated to a remote cache where either a remote hit or in the worst-case a remote miss followed by a remote storage read occurs. In IndiLog there is no distributed caching between compute nodes.

StateFun

StateFun [Sta] is built on top of Apache Flink [Fli] and provides a framework that combines stateful stream processing with serverless computing. Stateful functions get connected with each other to build event-driven applications, however, state and compute do not need to be physically co-located. Functions maintain and persist their state in the StateFun cluster but the code execution happens remotely and stateless on a FaaS platform, e.g., on AWS Lambda [Amab].

8.2 Distributed Shared Logs

Several distributed shared log designs have been proposed [Bal+13a; Bal+13b; JW21a; Wei+17; Loc+18; Din+20; Bal+a; Bal+b] and IndiLog borrows established concepts from them. Yet none of these systems focuses on the design and implications of the indexing architecture which is the main contribution of IndiLog . IndiLog borrows the high-throughput ordering protocol from Scalog [Din+20] and the metalog concept from Boki [JW21a] which itself is inspired from Delos [Bal+a]. Sub-streams in the log are used in Tango [Bal+13b], vCorfu [Wei+17] and Boki [JW21a].

CORFU

CORFU [Bal+13a] uses a distributed shared log over a flash cluster. Each log position is mapped to flash pages in the cluster. Clients that access CORFU use the mapping to directly write to and read from the address space of flash units. CORFU uses a dedicated sequencer to avoid clients from writing to the same log position. The sequencer maintains an increasing counter that corresponds to the log position of the tail. Clients contact the sequencer first to get tokens for next available positions they can use to write to. Holes in the log are created if clients obtain tokens but fail before writing to the log.

Tango

Tango [Bal+13b] presents a runtime for in-memory objects that are built from a distributed shared log. For each object the log durably stores the ordered history of modifications applied to the object. Clients replay the log to create objects and to synchronize their state. Clients store local views of objects in memory to avoid replaying the log from the beginning. Tango's runtime is built on top of CORFU. Tango introduces streams to counter replay overheads. A stream in Tango is a subset of all log records in the log. Records of a stream are logically connected by backpointers which are stored in log records. Backpointers allow clients to selectively read the log.

vCorfu

vCorfu [Wei+17] uses materialized streams to relieve the shared log from the burden of log playbacks for objects. When a client appends data to an object i.e., to a stream, the record is durably stored on the log and on a partition to which the stream belongs. This extends the append path as clients must send data to replicas of the log and to the replicas of a partition. However, reads are fast because a stream replica can playback all object modifications locally to create the object's recent state and send it to the client. vCorfu also introduces composable state machine replication (CSMR) for state efficiency: a base object may be composed of multiple smaller objects. These acquire smaller streams and induce less overhead when they are created and synchronized.

Scalog

Scalog [Din+20] introduces a new total ordering protocol. The log records are replicated first and global sequence numbers assigned afterwards. The storage nodes of each storage shard maintain a local order of their log records. They periodically send the progress of the local order to the ordering tier. The ordering tier, also periodically, determines for each storage shard the minimum progress all storage nodes within the shard made and send the results back to the storage nodes. The storage nodes then derive globally unique sequence numbers that are consistent with the local order. Since records are persisted first Scalog's design avoids holes in the log.

Delos

Delos [Bal+a] introduces the virtual log, a virtualization layer between applications and the log underneath. The virtual log exposes a transparent API to the applications. Virtualization allows the use of heterogeneous log implementations so-called Loglets without imposing any restrictions on the applications. Moreover, it separates two

concerns of shared logs: ordering and reconfiguration. The ordering of records is done by the Loglet whereas the virtual log handles reconfigurations. The virtual log can do so by sending a seal command to the Loglet which let the Loglet discard any new appends. When the Loglet is sealed the virtual log can safely reconfigure the Loglet (after a failure happened) or switch to another Loglet implementation.

Boki

Boki [JW21a] uses a distributed shared log as storage substrate to manage the state of serverless functions. It introduces the metalog concept to handle state transitions of the system and to guarantee the consistency of functions that may run concurrently on different compute nodes. Complete indexes maintained by the compute nodes allow ephemeral functions to locate records on the log. Boki adapts concepts from other state-of-the-art shared logs: Boki borrows the high-throughput protocol from Scalog [Din+20]. Sealing the metalog freezes any state transition and allows Boki to handle failures like in Delos [Bal+a]. Sub-streams for efficient reads introduced in Tango are adapted in Boki by using tags to identify streams. Tags are mapped to log sequence numbers. The mappings are stored in the complete indexes.

9 Conclusion

Distributed shared logs have been recently recognized as a promising substrate for serverless state management. Unfortunately, in this context, state-of-the-art fails to jointly meet two important requirements for serverless applications: accessing the log efficiently while maintaining unimpeded compute tier scalability. The cause is the reliance of complete indexes co-located with serverless functions on compute nodes. We showed that these complete indexes provide fast log access while severely limiting compute tier scalability and significantly increasing the risk of out-of-memory errors.

This master thesis also introduces IndiLog, a novel distributed indexing architecture for serverless applications to facilitate the efficient access to a distributed shared log. Using a combination of size-bound, optional local indexes alongside a sharded index tier which tackles the challenges of supporting log sub-streams and bounded reads, IndiLog obtains comparable or better performance to the state-of-the-art and, crucially for serverless workloads, without impeding compute tier scalability.

10 Future Work

In this master thesis we demonstrated the feasibility of distributed indexing on distributed shared logs. Still, there are open challenges and tasks which can be addressed in the future.

IndiLog introduces the Tag Cache to store the sequence numbers of customs tags. In the current design tags are only put into the Tag Cache from append operations. If a tag gets evicted and is read later, then, despite of an index miss, it is not put into the Tag Cache again. This is a problem for tags that do not satisfy temporal locality but are accessed frequently in the future. A solution to this problem would be a mechanism that detects popular tags for which lookups result in local index misses. The compute node may then retrieve the relevant index data from the index tier to fill its Tag Cache. Additionally, a LRU cache may support the Tag Cache: it stores results of index lookups that have led to misses in the Tag Cache before. Results must be static as a requirement for consistency i.e., the same bounded read for a tag always returns the same sequence number despite new log records that use the tag. The LRU cache can complement the Tag Cache for lookups that the Tag Cache never can serve, e.g., lookups for minimum sequence numbers of tags.

A remaining challenge is the synchronization between the index data structures and the record cache in a compute node. This is especially the case in which all variations of an index lookup for a record result in an index miss but the log record still lies in the cache.

Another problem is the coarse-grained locking of data structures that receive metalog updates. In shared logs that use a persist-first approach for new log records, the sequencer periodically propagates progress messages in a very small interval to keep the latencies of appends minimal. However, a smaller interval increases the update rate and thus the locking time. Consequently, reads are blocked more often from accessing the data structures. Reads would benefit from a finer-grained locking model that can still offer the same consistency guarantees. Furthermore, as seen in experiment 7.4 memory reallocation of index data structures lead to temporary throughput drops because append and read operations get blocked. Blocks can be reduced if memory allocation does not need to reallocate *old* memory and locks do not lock the entire index.

RAM in index nodes is sparse and index data are eventually persisted on disk.

However, disk reads significantly impact the performance of remote index lookups. Therefore, index nodes should distinguish between hot and cold index data based on temporal locality and accesses. Hot data is kept in memory or is reloaded from disk whereas cold data remains only on disk. The design of the index tier in IndiLog does not deal with this challenge yet.

Persistent memory may improve read latencies. It can be used as a memory extension for data structures in the compute nodes or as an intermediate storage medium between RAM and disk in the storage and index nodes.

List of Figures

List of Tables

Bibliography

[Amaa] Amazon. *Amazon DynamoDB: Fast, flexible NoSQL database service for single-digit millisecond performance at any scale*. https://aws.amazon.com/dynamodb/. Last accessed: June 26, 2022. Amazon.com, Inc.

[Amab] Amazon. *AWS Lambda - Run code without thinking about servers or clusters*. https://aws.amazon.com/lambda/. Last accessed: July 8, 2022. Amazon.com, Inc.

[Bal+a] M. Balakrishnan, J. Flinn, C. Shen, M. Dharamshi, A. Jafri, X. Shi, S. Ghosh, H. Hassan, A. Sagar, R. Shi, J. Liu, F. Gruszczynski, X. Zhang, H. Hoang, A. Yossef, F. Richard, and Y. J. Song. "Virtual Consensus in Delos." In: *OSDI 2020*.

[Bal+b] M. Balakrishnan, C. Shen, A. Jafri, S. Mapara, D. Geraghty, J. Flinn, V. Venkat, I. Nedelchev, S. Ghosh, M. Dharamshi, J. Liu, F. Gruszczynski, J. Li, R. Tibrewal, A. Zaveri, R. Nagar, A. Yossef, F. Richard, and Y. J. Song. "Log-Structured Protocols in Delos." In: *SOSP 2021*.

[Bal+13a] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. "CORFU: A Distributed Shared Log." In: *ACM Trans. Comput. Syst.* 31.4 (Dec. 2013). ISSN: 0734-2071. DOI: 10.1145/2535930.

[Bal+13b] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. "Tango: Distributed Data Structures over a Shared Log." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 325–340. ISBN: 9781450323888. DOI: 10.1145/2517349.2522732.

[Del] C. Delimitrou. *DeathStarBench*. https://github.com/delimitrou/DeathStarBench. Last accessed: June 30, 2022. Systems Architecture and Infrastructure Lab (SAIL), Cornell University.

[Din+20] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. Van Renesse. "Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log." In: *Proceedings of the 17th Usenix Conference on Networked Systems Design*

and Implementation. USA: USENIX Association, 2020, pp. 325–338. ISBN: 9781939133137.

[Fli] Flink. *Apache Flink — Stateful Computations over Data Streams*. `https://flink.apache.org/`. Last accessed: June 27, 2022. Apache Software Foundation.

[Goo] Google. *Abseil*. `https://abseil.io/`. Last accessed: July 6, 2022. Google LLC.

[Hir] M. Hirabayashi. *Tkrzw: a set of implementations of DBM*. `https://dbmx.net/tkrzw/`. Last accessed: June 21, 2022.

[Int] Intel. *oneAPI Threading Building Blocks (oneTBB)*. `https://oneapi-src.github.io/oneTBB/`. Last accessed: June 21, 2022. Intel Organization.

[JW21a] Z. Jia and E. Witchel. "Boki: Stateful Serverless Computing with Shared Logs." In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 691–707. ISBN: 9781450387095. DOI: 10.1145/3477132.3483541.

[JW21b] Z. Jia and E. Witchel. "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 152–166. ISBN: 9781450383172. DOI: 10.1145/3445814.3446701.

[Kli+] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics." In: *OSDI 2018*.

[KYK] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. "Centralized Core-Granular Scheduling for Serverless Functions." In: *SoCC 2019*.

[Loc+18] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan. "The Fuzzylog: A Partially Ordered Shared Log." In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 357–372. ISBN: 9781931971478.

[Met] Meta. *RocksDB – A persistent key-value store for fast storage environments*. `http://rocksdb.org/`. Last accessed: June 21, 2022. Meta Platforms, Inc.

[Poc] D. Poccia. *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/`. Last accessed: June 21, 2022. Amazon.com, Inc.

[PVS] Q. Pu, S. Venkataraman, and I. Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure." In: *NSDI 2019*.

[Rom+] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini. "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications." In: *SoCC 2021*.

[Sch+21] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson. "What Serverless Computing is and Should Become: The next Phase of Cloud Computing." In: *Commun. ACM* 64.5 (Apr. 2021), pp. 76–84. ISSN: 0001-0782. DOI: `10.1145/3406011`.

[Sha+20] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider." In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[Sre+20] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. "Cloudburst: Stateful Functions-as-a-Service." In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097. DOI: `10.14778/3407790.3407836`.

[Sta] StateFun. *Stateful Functions: A Platform-Independent Stateful Serverless Stack*. `https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/`. Last accessed: June 27, 2022. Apache Software Foundation.

[Wei+17] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi. "VCorfu: A Cloud-Scale Object Store on a Shared Log." In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, pp. 35–49. ISBN: 9781931971379.

[Wu+] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. "Anna: A KVS for Any Scale." In: *ICDE 2018*.

[Zha+] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. "Fault-tolerant and transactional stateful serverless workflows." In: *OSDI 2020*.

[Zoo] Zookeeper. *Welcome to Apache ZooKeeper: Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.* `https://zookeeper.apache.org/`. Last accessed: June 21, 2022. Apache Software Foundation.