IndiLog

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

Maximilian Wiesholler
Advisor: Dr. Florin Dinu
Supervisor: Prof. Dr.-Ing. Pramod Bhatotia
Chair of Decentralized Systems Engineering
https://dse.in.tum.de/

15.01.2022 – 15.07.2022

Motivation: research context

- Serverless functions
  - Ephemeral nature
  - Functions rely on service for state management

- Cloud storage services perform bad for state-sharing
  - May not offer consistency guarantees
  - Performance and costs trade-offs

State-of-the-art

- Recent development of new systems to improve state management
  - Cloudburst[1], Pocket[2], Faa$t[3], …
- Boki: distributed shared logs as a promising serverless storage substrate[4]
  - Failure resilience
  - Consistency guarantees

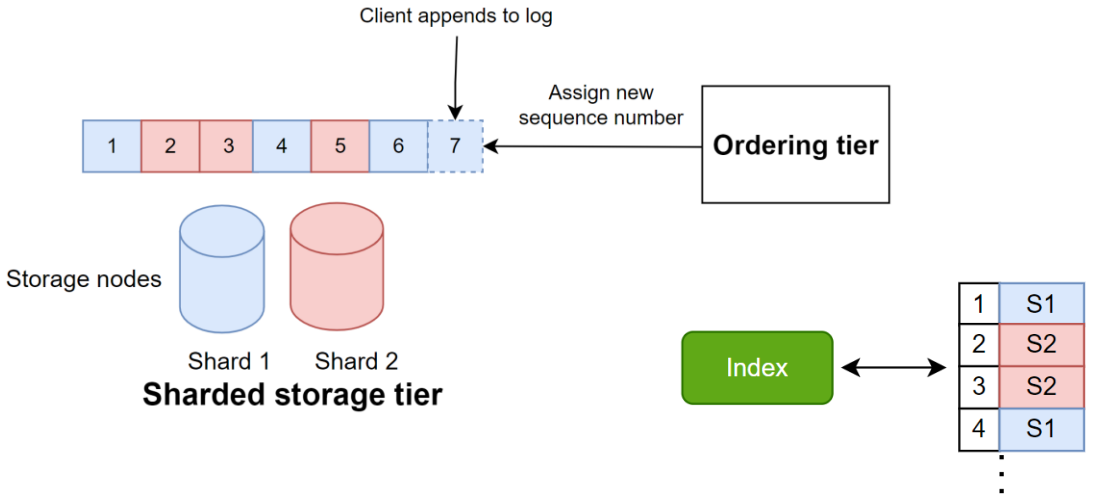
[1]V. Sreekanti et al. “Cloudburst: Stateful Functions-as-a-Service.”
[2]A. Klimovic et al. “Pocket: Elastic Ephemeral Storage for Serverless Analytics.”
[3]F. Romero et al. “Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications.”
[4]Z. Jia et al. “Boki: Stateful Serverless Computing with Shared Logs.”

Background: distributed shared logs

Log: sequence of immutable log records; append-only

2 main tiers:

storage and ordering



For serverless

- Compute tier to run functions
- Indexes to locate records on the log

Research gap

Distributed shared logs for serverless state management shift **indexing** into the focus

The current approach to indexing has the following limitations:

- Functions share resources with indexes
- Scalability of the compute tier is impeded by the design of indexing

Problem statement

How to design an efficient indexing architecture

- Do not impede the scaling of the compute tier by the design of indexing
- Limit resources for indexes co-located with functions on compute nodes

IndiLog:
a distributed indexing architecture for shared logs

Compute nodes maintain optional, local and incomplete indexes
A sharded index tier balances the index data across index nodes and is complete

System design goals:

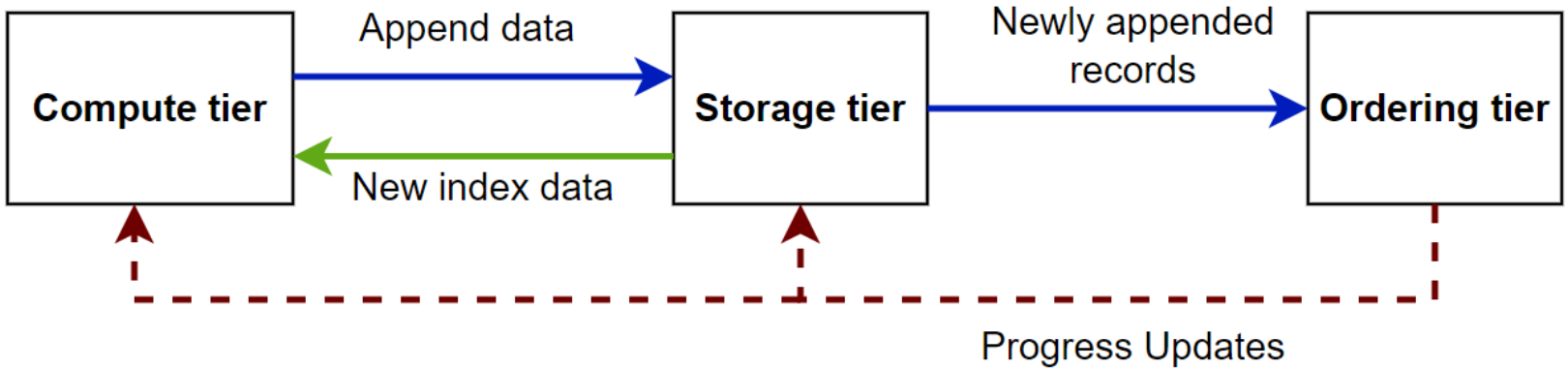
Performance:	many index lookups are captured locally
Resource efficiency:	local indexes are size-bounded
Scalability:	compute tier scalability is not impeded
Functions run anywhere:	no constraints where functions run

Outline

- ~~Motivation 1: high-level~~
- Background
  - Boki
- Motivation 2: Boki's limitations
- Design: IndiLog
- Implementation
- Evaluation

Boki

State-of-the-art distributed shared log for state management of serverless functions[1]

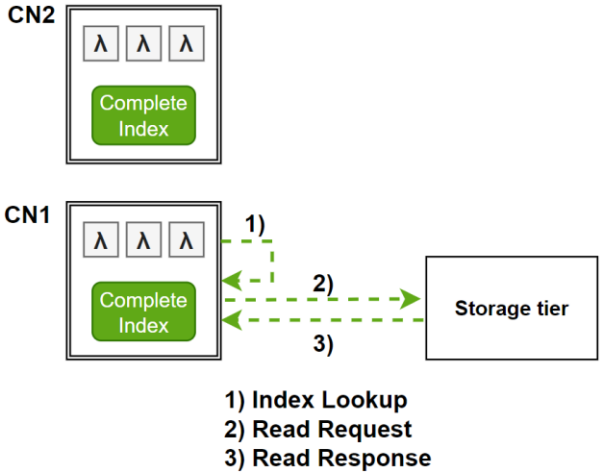


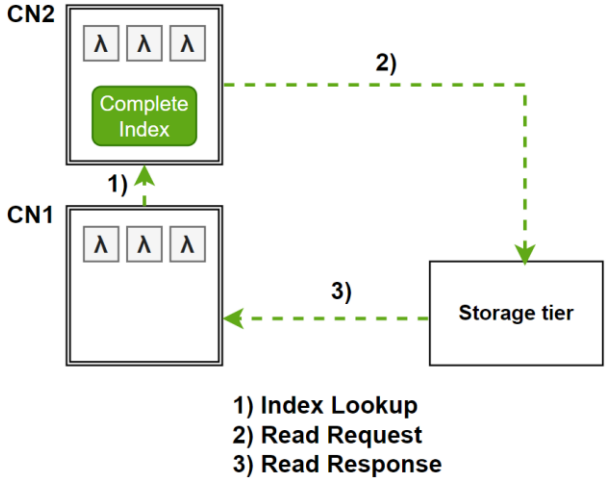
[1]Z. Jia et al. "Boki: Stateful Serverless Computing with Shared Logs."

Boki: indexing

Indexes are complete and co-located with serverless functions on compute nodes

The complete index lets functions locate records on the storage tier





Boki: read semantics

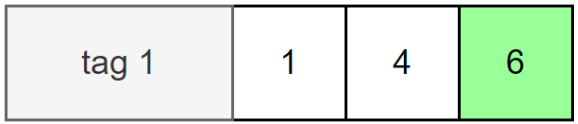
Boki uses **tags** to create logical sub-streams over the log[1]

- Function needs only reading records that belong to the sub-streams

tag 1	1	4	6		
tag 2	2	3	8	9	10
tag 3	5	7			

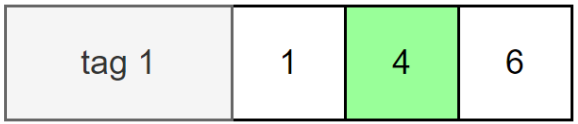
Boki uses **bounded reads** which may not target a specific sequence number

Read tag 1 with **6 ≤ X**



6 is **exact** match

Read tag 1 with **5 ≥ X**



4 is **closest** match

Exact match
Bound is in sub-stream

Closest match
Bound is not in sub-stream

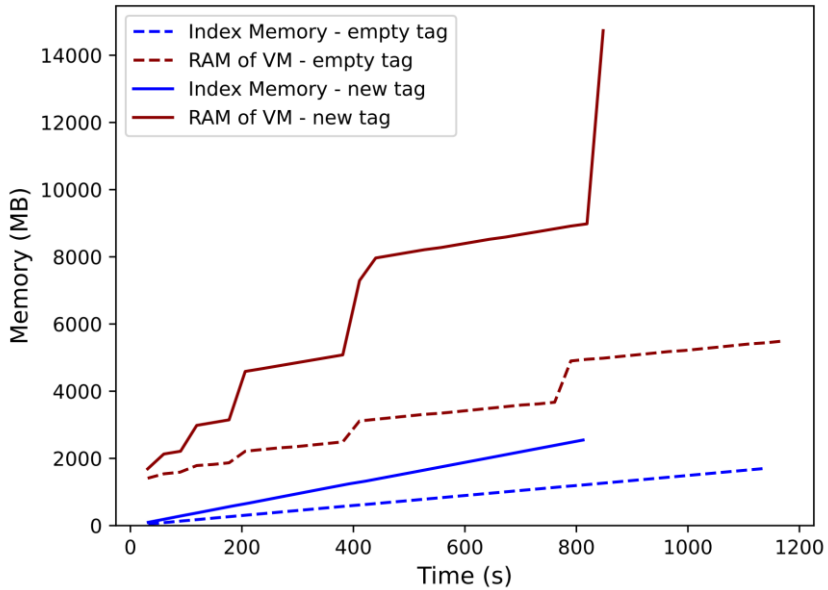
[1]M. Balakrishnan et al. "Tango: Distributed Data Structures over a Shared Log."

Outline

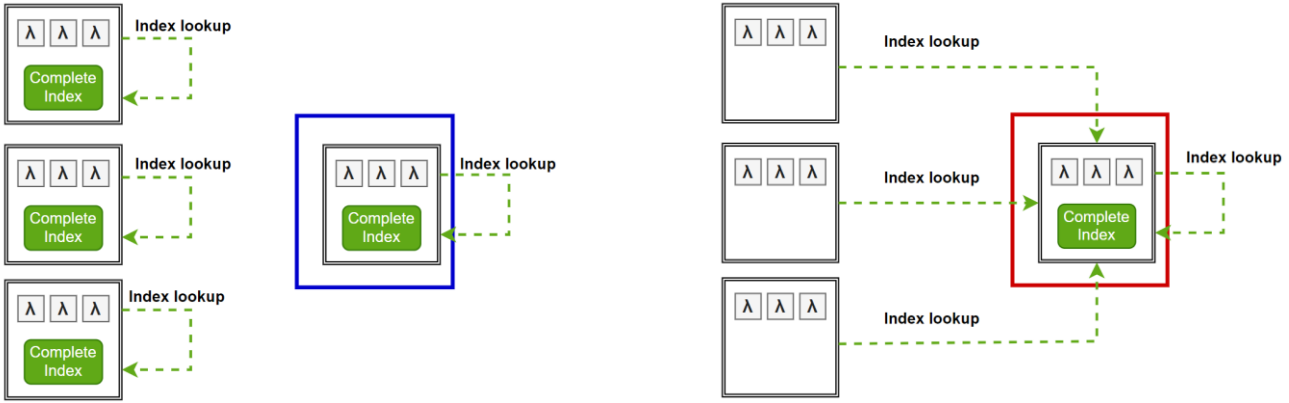
- ~~Motivation 1: high-level~~
- ~~Background~~
- Motivation 2: Boki's limitations
- Design
- Implementation
- Evaluation

Boki: out-of-memory crashes in compute nodes

Memory usage of indexes increase over time and the compute nodes eventually crash



Boki: index lookups add high contention after scaling



Increase workload

Throughput [kOp/s]	40.73	60.44	71.48	77.11	93.09	100.82	102.9
Throughput [kOp/s]	33.84	46.28	52.93	56.66	63.65	63.28	63.78
Ratio [%]	83.1	76.6	74.0	72.1	68.4	62.8	62.0

Outline

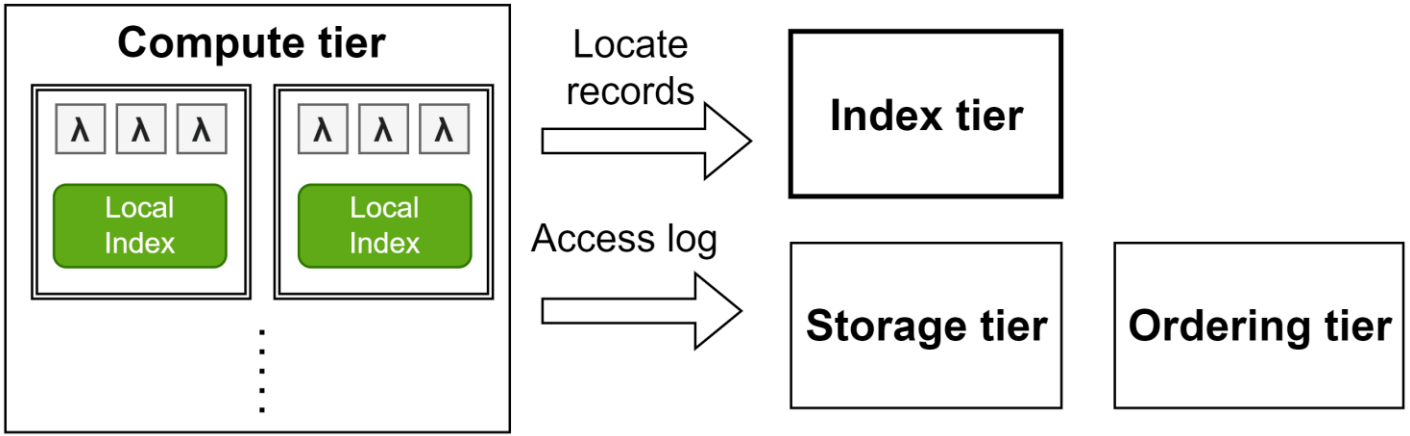
- ~~Motivation 1: high-level~~
- ~~Background~~
- ~~Motivation 2: Boki's limitations~~
- Design: IndiLog
- Implementation
- Evaluation

IndiLog: system overview

4 tiers: compute, ordering, storage and **index**

Compute nodes have optional, **local indexes** to capture most of the index lookups

- size-limited by eviction policies

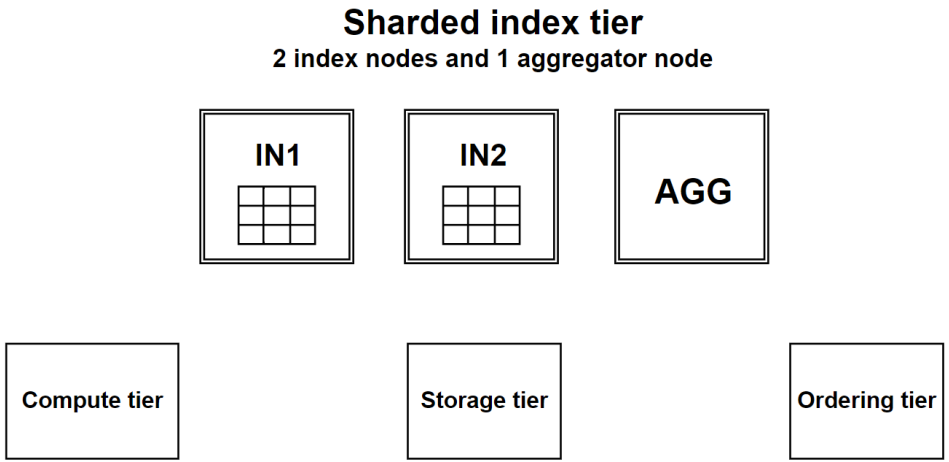


Design of the index tier

Sharded: indexes on any two index shards do not intersect

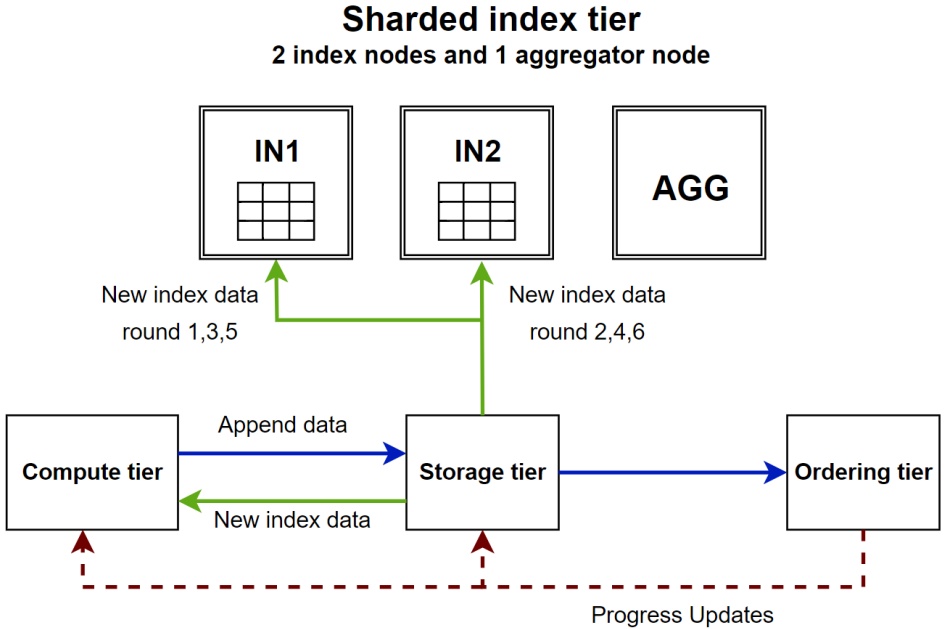
Completeness: the index tier can serve all index lookups

Aggregating: aggregator determines the closest match from index lookups



Append path

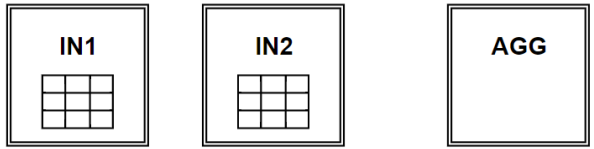
New index data is sent to the index tier in round-robin fashion to balance the distribution



Read path

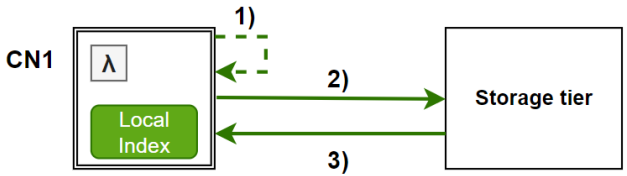
In IndiLog we have three types of reads

Type 1: the local index of a compute node has a match for the index lookup



No lookup in index tier:

2 * one-way network latency



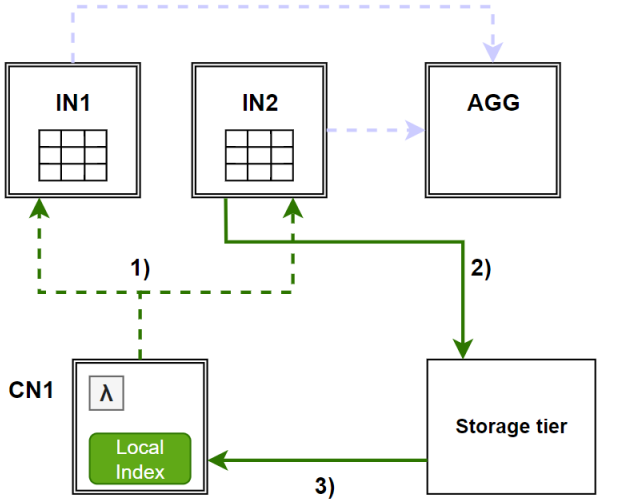
1) Index lookup with local hit
2) Read request
3) Read response

Read path

Type 2: the index lookup goes to the index tier – one index shard has an **exact match**

Index node with exact match can safely forward read request:

3 * one-way network latency

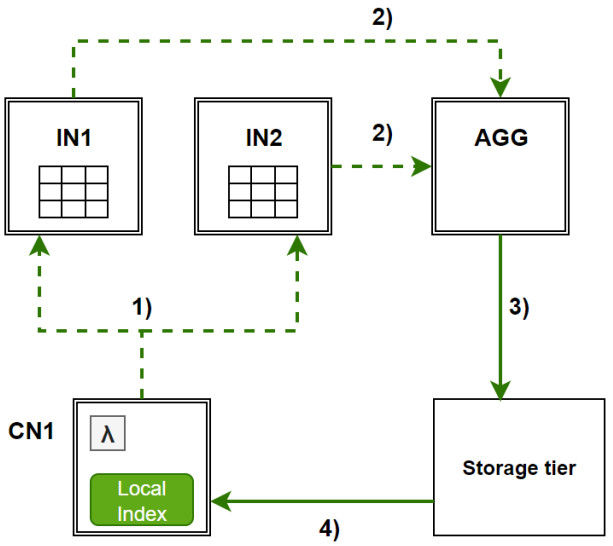


1) Index tier lookup
2) Read request from IN2 with exact match
3) Read response

Read path

Type 3: the index lookup goes to the index tier – all index shards have **closest match**

Aggregator **aggregates the best match**



1) Index tier lookup
2) Closest matches from index nodes
3) Read request for aggregated best match
4) Read response

Aggregator forwards read request:
4 * one-way network latency

Outline

- ~~Motivation 1: high-level~~
- ~~Background~~
- ~~Motivation 2: Boki's limitations~~
- ~~Design: IndiLog~~
- Implementation
- Evaluation

Implementation

IndiLog is built in C++ by adding an indexing architecture on top of Boki

- Index data structures implemented with the C++ Standard library and Abseil libraries[1]
- Index data structures use a coarse-grained locking
- Aggregator uses fine-grained, per-request locks (data structures from oneTBB[2])
- ZooKeeper[3] for configuration and dynamical scaling of the compute tier

IndiLog: 7,892 LoC | **Benchmarks:** 19,131 LoC

[1]Abseil libraries: https://abseil.io/
[2]Threading Building Blocks (oneTBB): https://oneapi-src.github.io/oneTBB/
[3]Apache ZooKeeper: https://zookeeper.apache.org/

Outline

- ~~Motivation 1: high-level~~
- ~~Background~~
- ~~Motivation 2: Boki's limitations~~
- ~~Design: IndiLog~~
- ~~Implementation~~
- Evaluation

Evaluation

- What effects do we observe when we scale the compute tier of IndiLog?
  - Different workloads, various scaling sizes
- How does the index tier of IndiLog perform?
  - Type 2 and type 3 reads
- How does IndiLog behaves for real applications?
  - Object storage used on top of IndiLog

Evaluation

- Experimental setup:
  - Cloud VMs: RAM 16GB | vCPUs 4 | Ubuntu 20.04 | Kernel version 5.10.0
  - Latency: 180 µs +/- 40 µs Bandwidth: 2,127 Mbps
- Workload: append new records (1 KB) and mix reads for persisted records

We compare IndiLog against Boki

	IndiLog	Boki
Compute Tier	4 VMs	4 VMs
Storage Tier	4 VMs	4 VMs
Ordering Tier	3 VMs	3 VMs
Index Tier	3 VMs (2 IN, 1 AGG)	-
Local Index	20 MB	complete i.e.,16 GB

Scaling the compute tier from 1 to 4

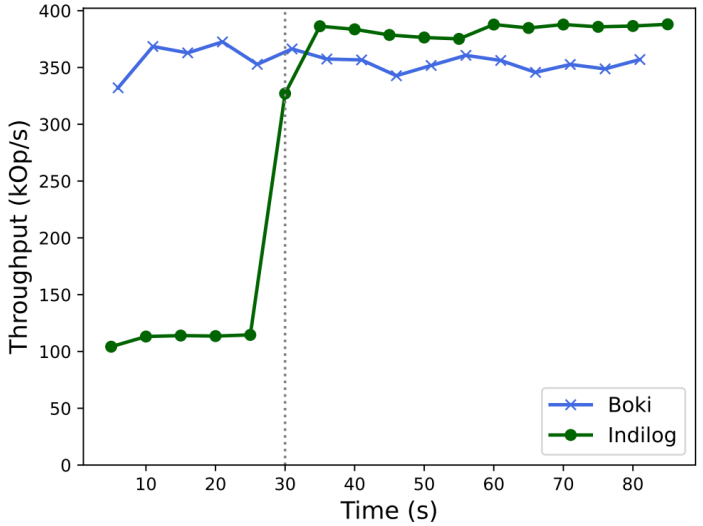
Workload: 50/50 Append/Read + 87% Local Index Hit Ratio in IndiLog

Indilog
- Starts with 1 compute node
- Scales to 4 compute nodes after 30 sec

Boki
- Cannot scale dynamically
- Configured as scaling happened

→ only 1 compute node has a complete index

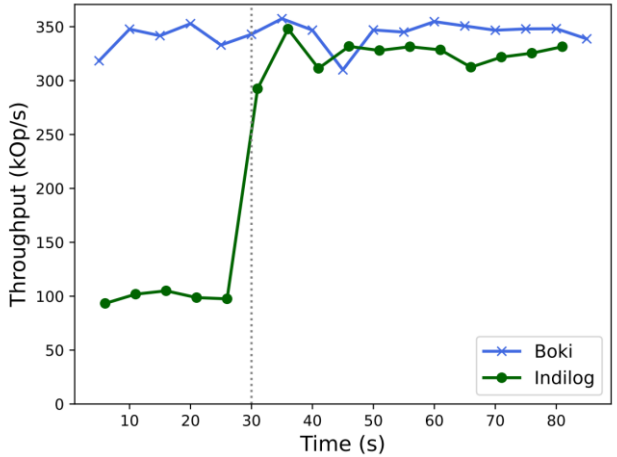
→ 3 compute nodes send remote index lookups



IndiLog beats Boki when IndiLog captures many index lookups locally

Scaling the compute tier from 1 to 4

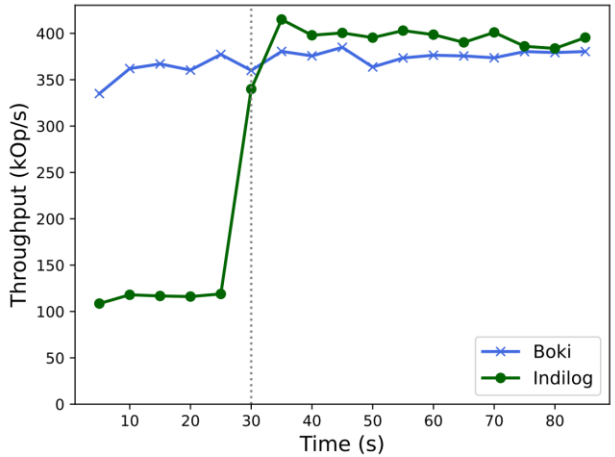
Workload: 50/50 Append/Read + 20% Local Index Hit Ratio in IndiLog



A low index hit ratio in IndiLog lowers the overall throughput

Scaling the compute tier from 1 to 4

Workload: 5/95 Append/Read + 20% Local Index Hit Ratio in IndiLog

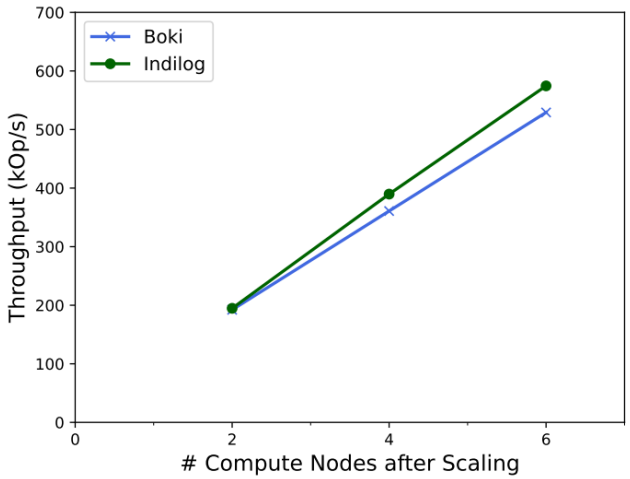


A read-heavy workload favors IndiLog:
Fewer write updates on index data structures

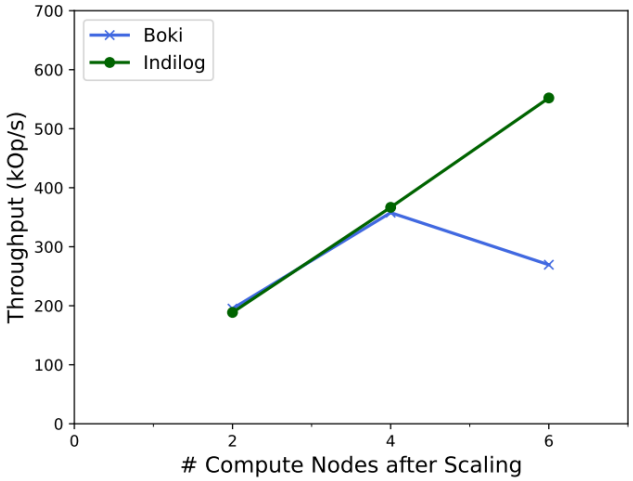
Scaling the compute tier from 1 to 2/4/6

87% Local Index Hit Ratio in IndiLog

50/50 Append/Read



5/95 Append/Read

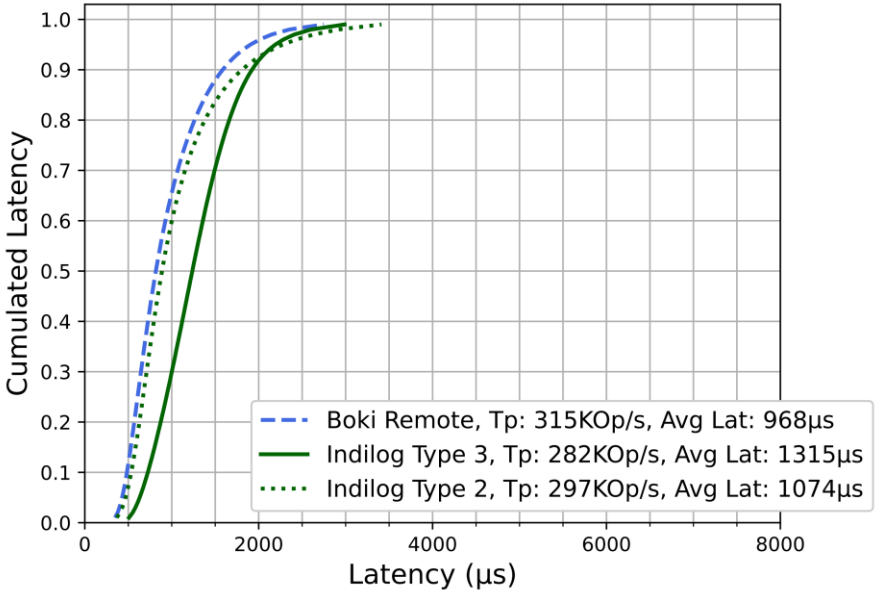


Boki's node with the complete index gets under heavy contention

Read latencies of the index tier

IndiLog
Local index disabled
- All index lookups go the index tier

Boki
- Few compute nodes maintain complete indexes but do not run functions
- Compute nodes with functions do remote index lookups only



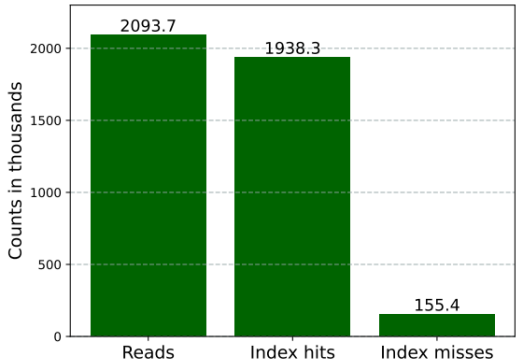
IndiLog‘s sharded index tier comparable to remote complete indexes

Real application: object storage

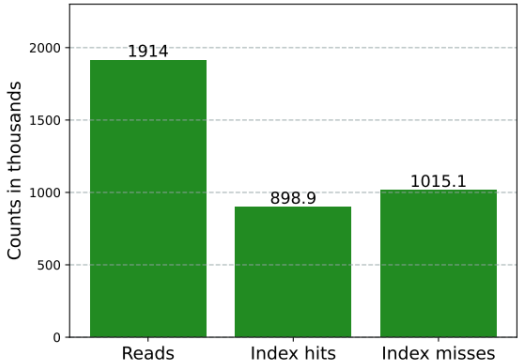
IndiLog as infrastructure layer of an object storage library with transaction support

- Workload: functions doing CRUD operations on key-value objects

Default index size (~20 MB)



Small index size (~0.2 MB)



System	Throughput [Op/s]
Boki (complete index)	8950.9
IndiLog (default)	8700.1
IndiLog (small)	8381.4

IndiLog is able to capture 93% of index lookups locally

Even with a very small index IndiLog captures almost 50% of lookups locally

Summary

Current state-of-the-art shared logs neglect efficient indexing

- Boki's complete index:
  - Leads to high RAM consumption and eventually OOM crash
  - Impedes scalability of the compute tier

IndiLog

- Local indexes + index tier for efficient indexing of log records
- Dynamic scaling of the compute tier

IndiLog source code: https://github.com/MaxWies/IndiLog
Benchmark source code: https://github.com/MaxWies/indilog-benchmarks

Discussion & Questions

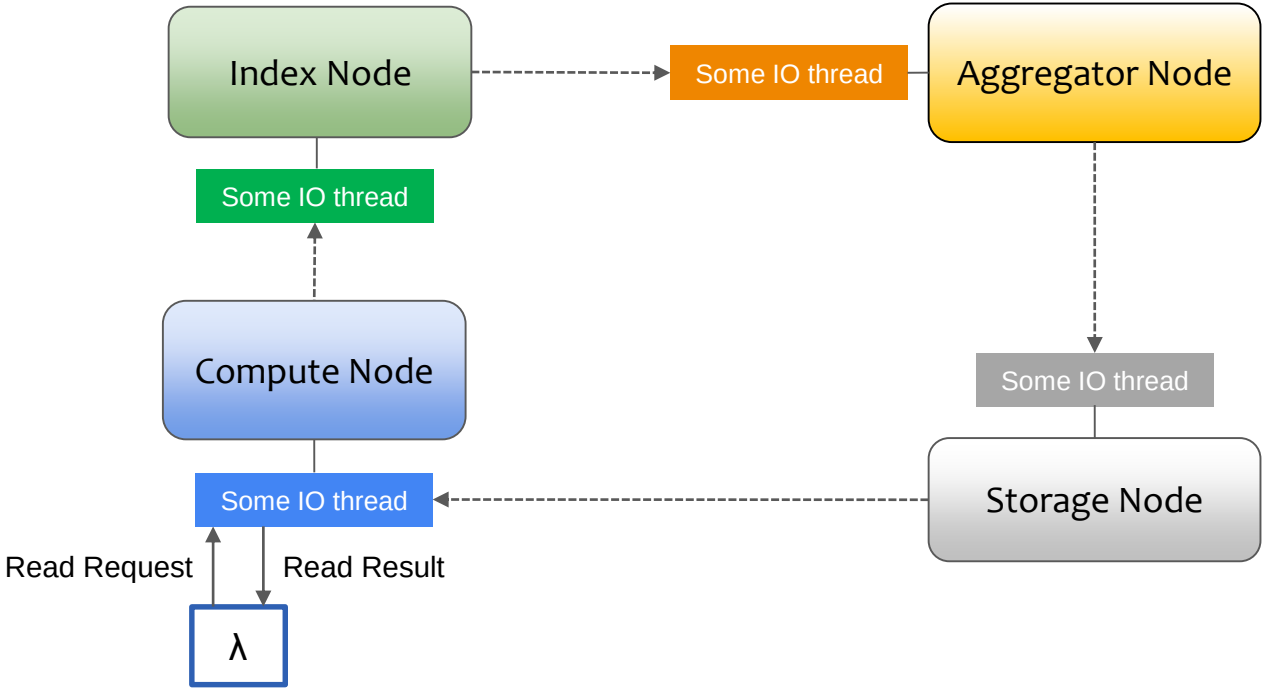
Backup

Serverless computing

Fine-grained functions run developers code on a Function-as-a-Service (FaaS) platform

Configuration and management of the platform is done by the platform provider

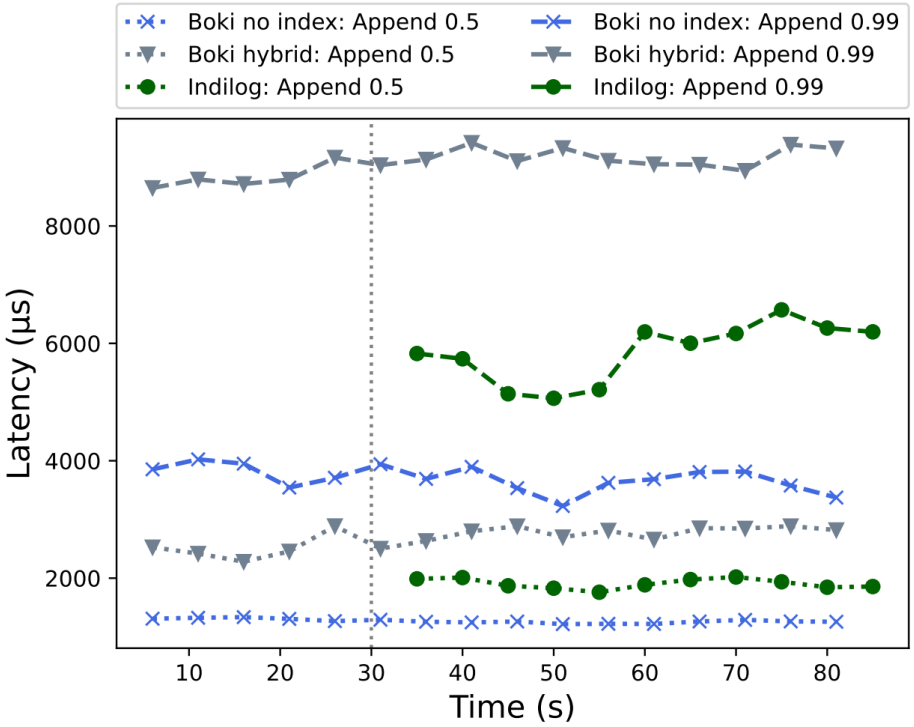
Threading model for reads (here: type 3)



Function waits until it receives result - Nodes process requests asynchronously

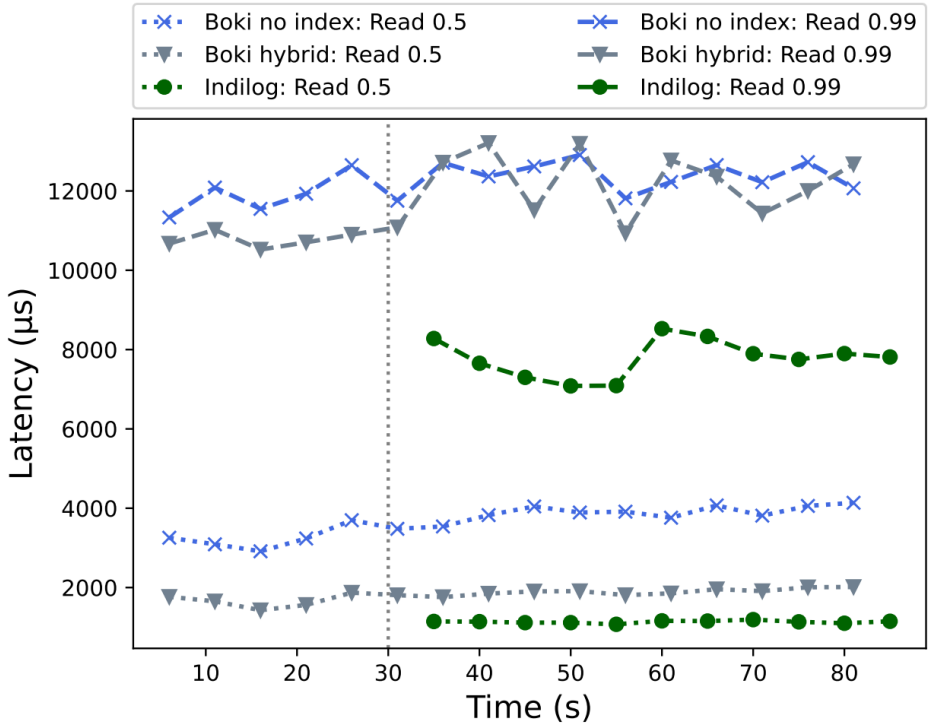
Scaling the compute tier from 1 to 4

50/50 Append/Read + 87% Local Index Hit Ratio in IndiLog

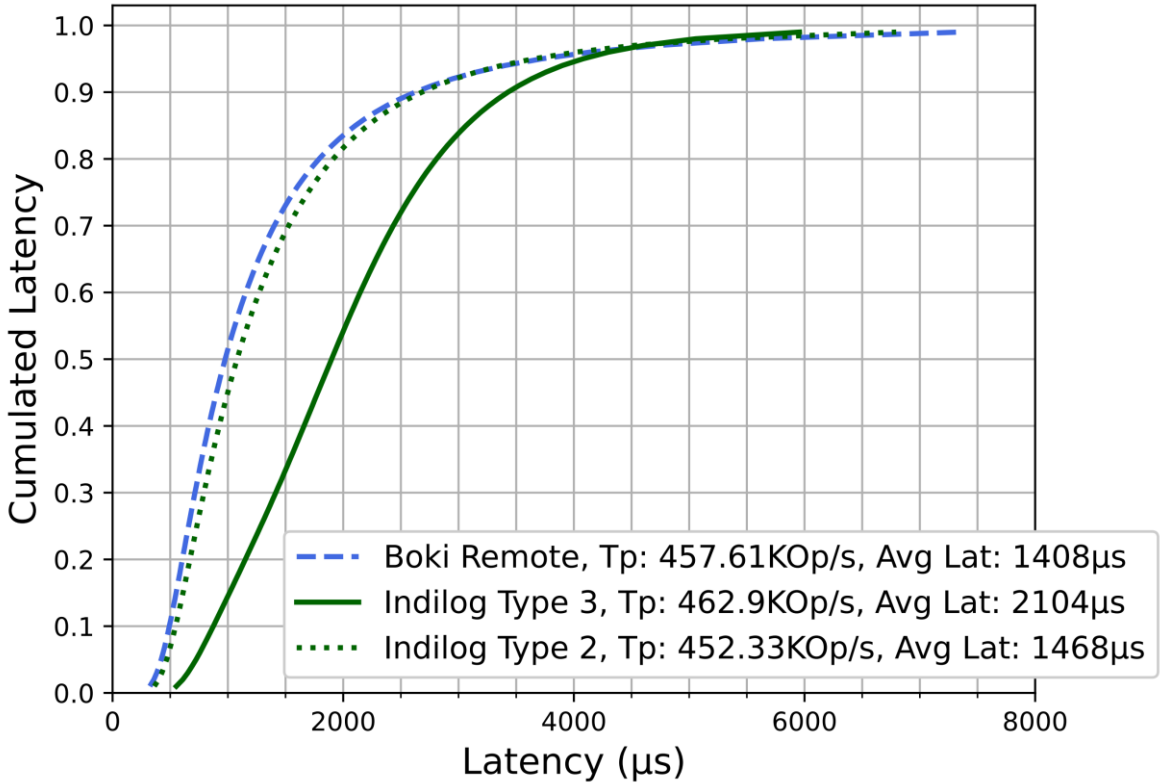


Scaling the compute tier from 1 to 4

50/50 Append/Read + 87% Local Index Hit Ratio in IndiLog



Read latencies of the index tier (high concurrency)



Scaling the index tier

Type 3 reads - all index lookups go to the index tier

S: Index **s**hards

R: **R**eplication factor

A: **A**ggregators

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

Real application: object store

Throughput Comparison

System	Configuration	Throughput [Op/s]
Boki	4 compute nodes with complete indexes	8950.9
IndiLog	4 compute nodes, 20 MB local index	8700.1
IndiLog	4 compute nodes, 0.2 MB local index	8430.5
Boki	4 compute nodes with no indexes 2 compute nodes with complete indexes but no functions	8381.4