IndiLog

Bridging Scalability and Performance in Stateful Serverless Computing with Shared Logs

Maximilian Wiesholler
TU Munich / Huawei Research Center Munich

Florin Dinu
Huawei Research Center Munich

Javier Picorel
Huawei Research Center Munich

Pramod Bhatotia
TU Munich

HUAWEI

Motivation: Serverless Computing

- **Serverless functions are stateless by design**
  - However: non-trivial applications need statefulness
  - How to get statefulness? Rely on a dedicated storage service for state management

- **Serverless functions run short-lived tasks**
  - Rely on fast storage access

- **Cloud storage services (e.g., S3) 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 for serverless functions

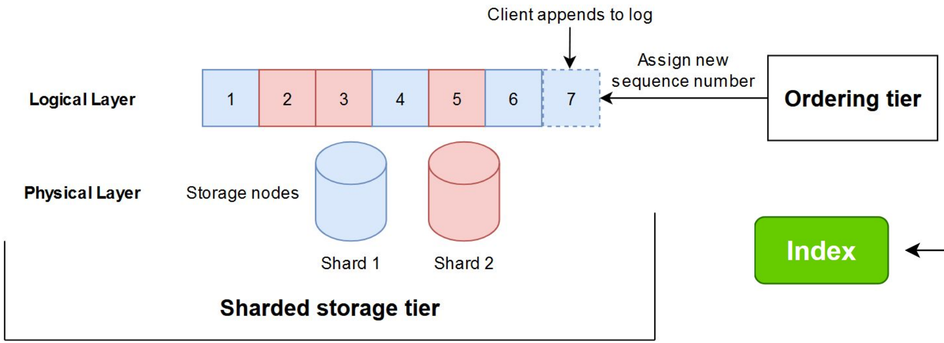
- **Boki [SOSP21][1]:** distributed shared logs – a promising serverless storage substrate to manage function state
  - Resilient to failures
  - Offers consistency guarantees

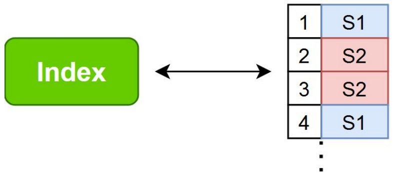
[1]Z. Jia et al. “Boki: Stateful Serverless Computing with Shared Logs.”

Background: distributed shared logs

Log: sequence of immutable log records; append-only

2 main tiers:
storage and ordering



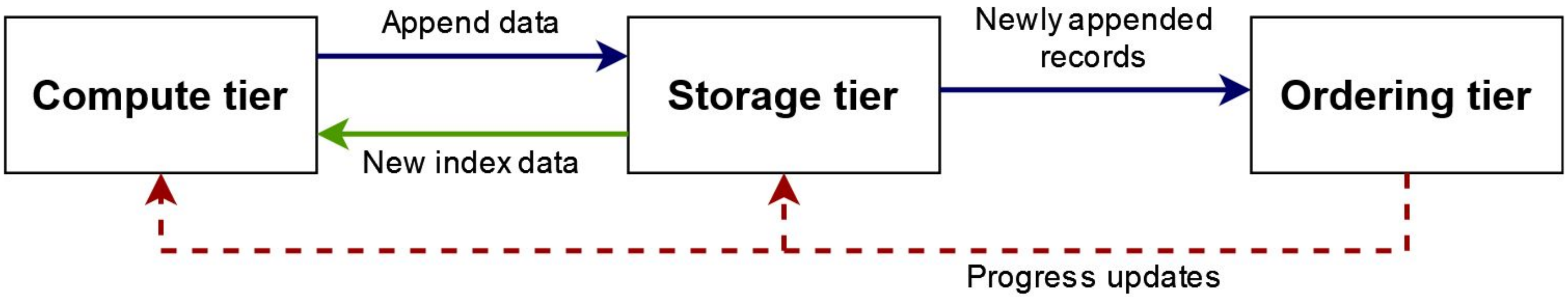


For serverless

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

Boki: distributed shared log

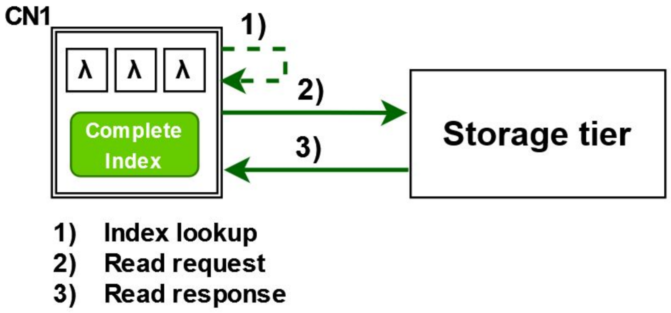
State-of-the-art for state management of serverless functions

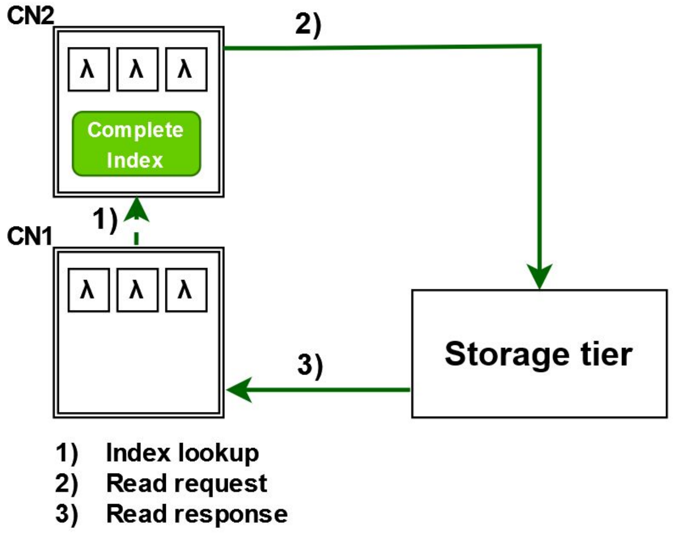


Boki: indexing

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

The complete index lets functions locate records on the storage tier.

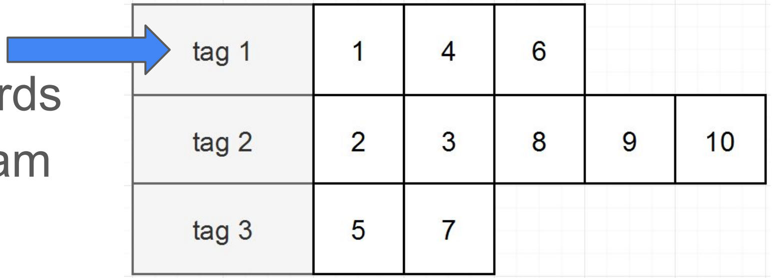




Boki: read semantics

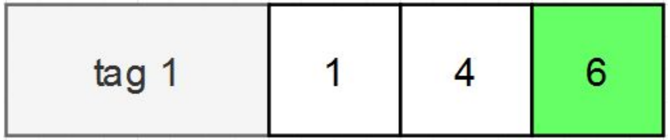
Boki uses **tags** to create logical sub-streams over the log[1]

- Function only needs to read records that belong to the same sub-stream



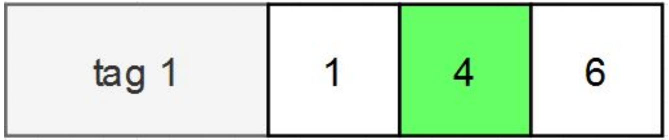
Boki uses **bounded reads** which may not target a specific sequence number

Read **tag 1** with **X ≥ 6**



6 is **exact** match

Read **tag 1** with **X ≤ 5**



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."

Research gap

Shared logs for serverless state management bring indexing in focus.

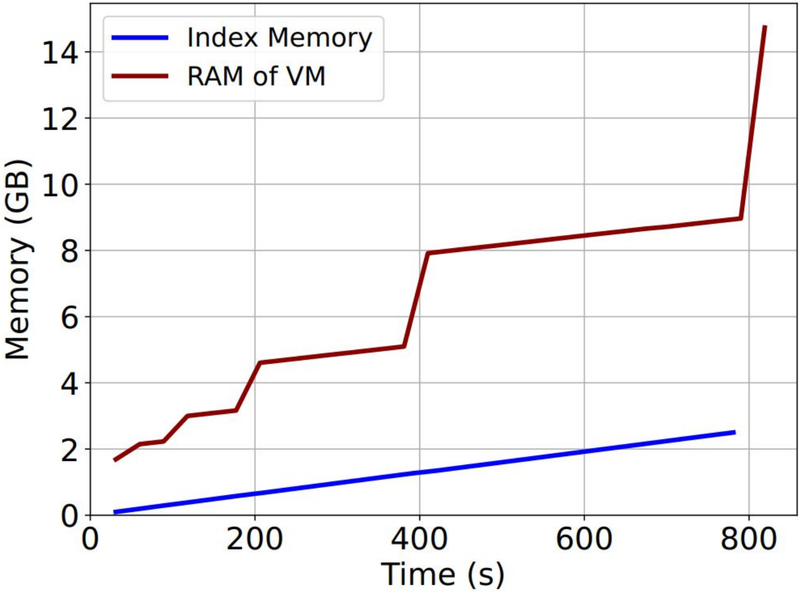
Limitations of the current approach to indexing:

- Functions share resources with co-located indexes
- Scalability of the compute tier is impeded by the design of indexing
- Indexes can exceed local resources

Boki: complete index quickly exceeds memory limit

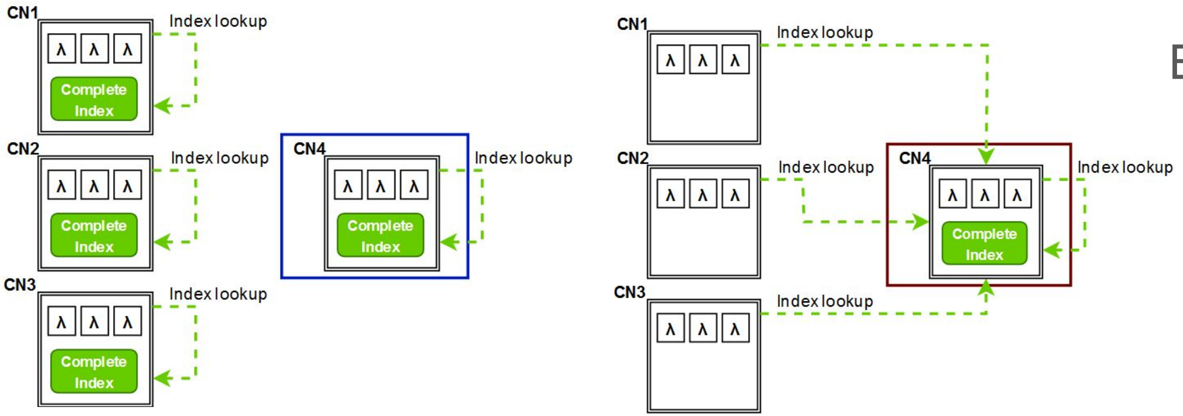
Experiment: continuously append new records to the log to create new index data

- Measure the memory of the compute node



Index memory usage increases over time and eventually causes OOM

Boki: index lookups add high contention after scaling



Experiment:
- Functions in each CN: append records and read them
- Increase workload
- Measure throughput of **CN4**

Increase workload →

Throughput [kOp/s] @ CN4	40.73	60.44	71.48	77.11	93.09	100.82	102.9
Throughput [kOp/s] @ CN4	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

Red **CN4**: Significant performance drop due to index lookup contention

Problem statement

How to design an efficient indexing architecture for a distributed shared log

- 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

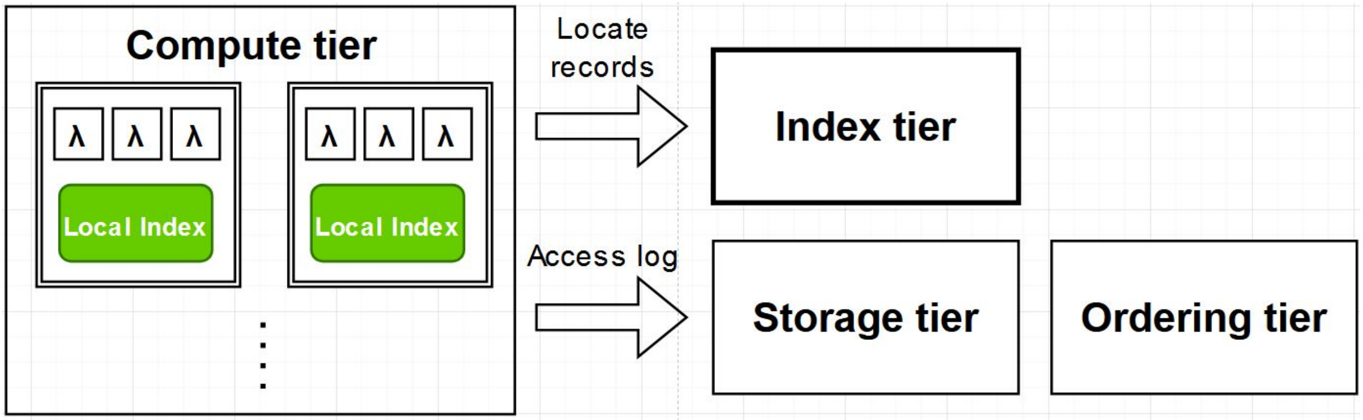
Performance:	many index lookups are captured locally
Resource efficiency:	local indexes are size-bounded
Scalability:	compute tier scalability is not impeded
Functions run anywhere:	no constraints where functions run

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

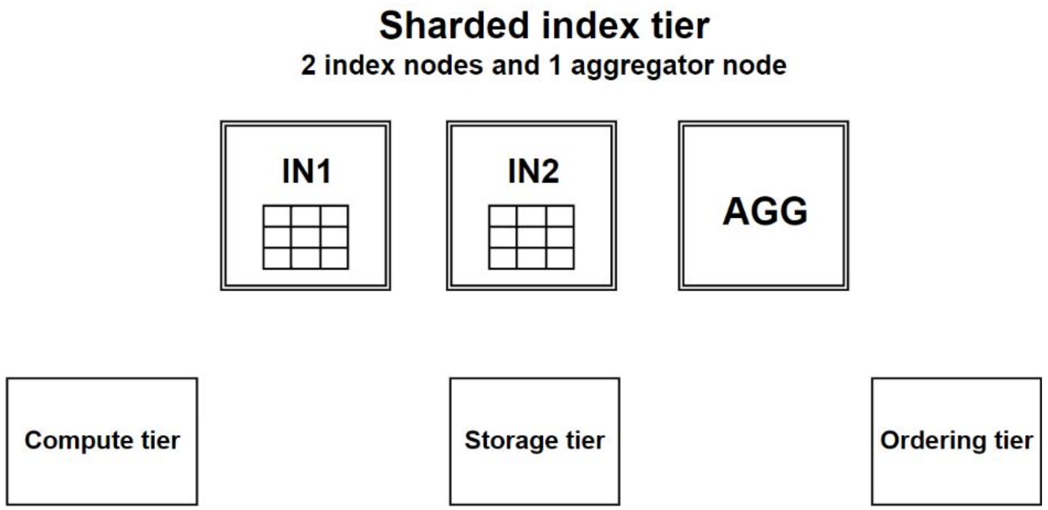


Design of the index tier

Sharded: indexes on any two index shards do not intersect

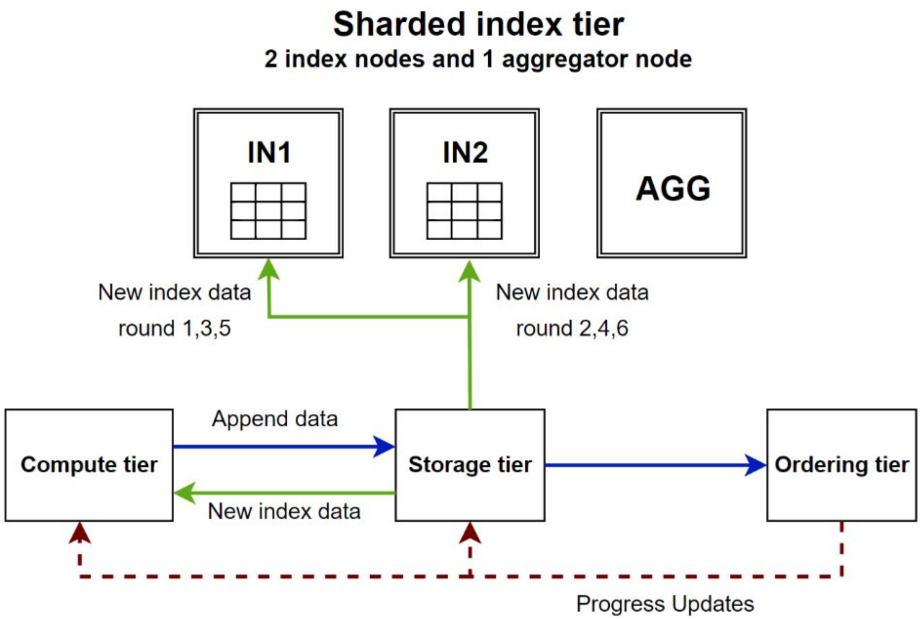
Completeness: the index tier can serve all index lookups

Aggregating: aggregator determines the closest match from index lookups



Append path

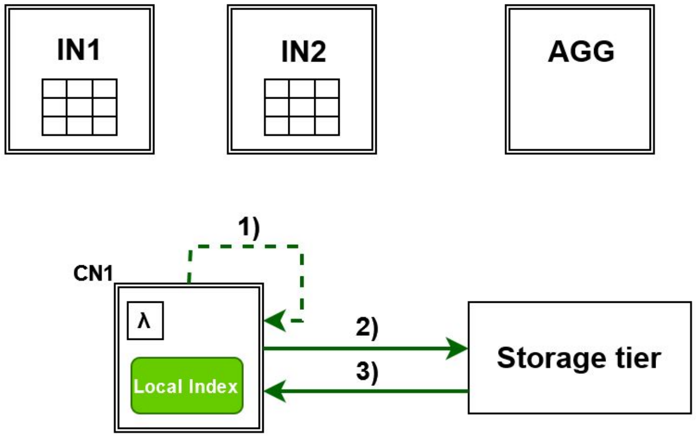
New index data is balanced (round-robin) over the index tier nodes



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



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

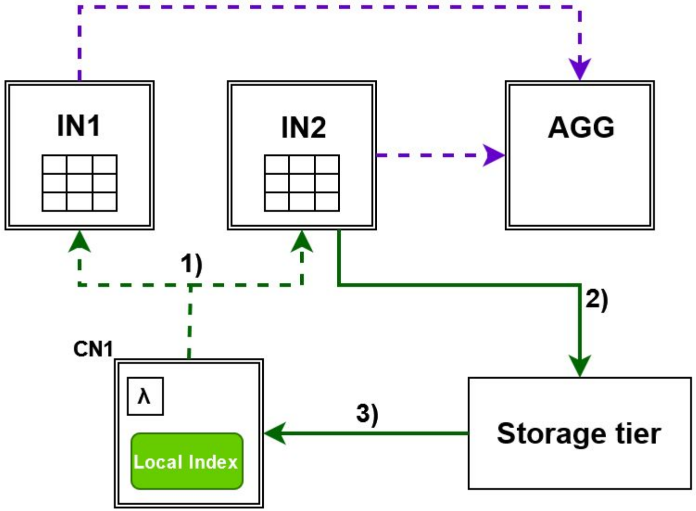
No lookup in index tier:

2 * one-way network latency

Read path

In IndiLog we have three types of reads

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



1. Index tier lookup
2. IN2 with exact match → IN2 read request
3. Read response

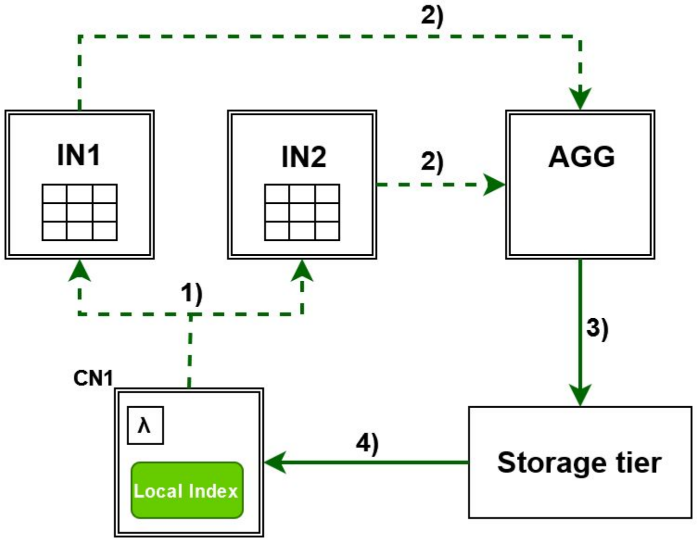
Index node with exact match forwards read request:

3 * one-way network latency

Read path

In IndiLog we have three types of reads

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



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

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

Evaluation

For IndiLog we want to observe

- … the effects of scaling its compute tier
- … the performance of the index tier
- … its behavior for real applications

Evaluation: Setup

- Cloud VMs with **16 GB RAM** and **4 vCPUs**
- Workload: mix of appending records (1 KB) and reading 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

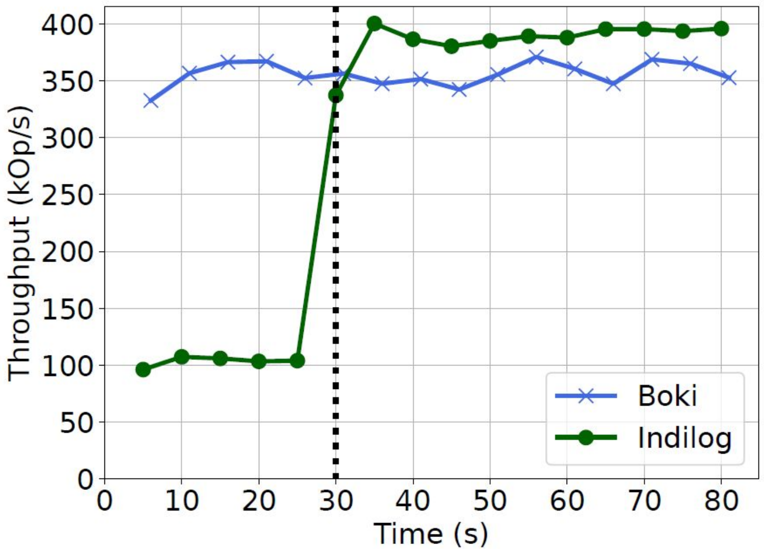
IndiLog

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

Boki

- Cannot scale dynamically
- Simulate past scaling event:

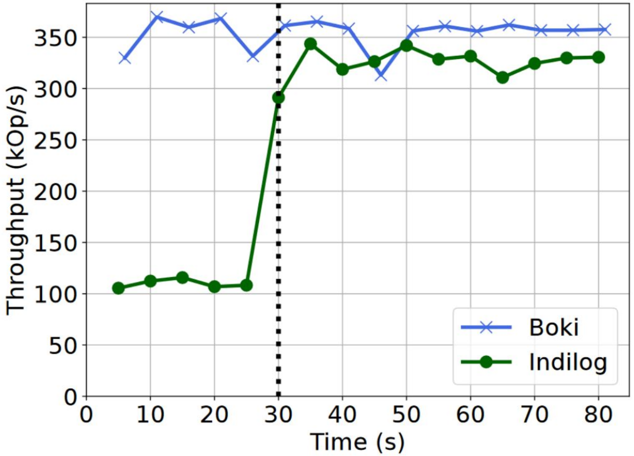
→ only 1 compute node has a complete index
→ 3 compute nodes send remote index lookups



IndiLog beats Boki when IndiLog captures many index lookups locally

Scaling the compute tier from 1 to 4

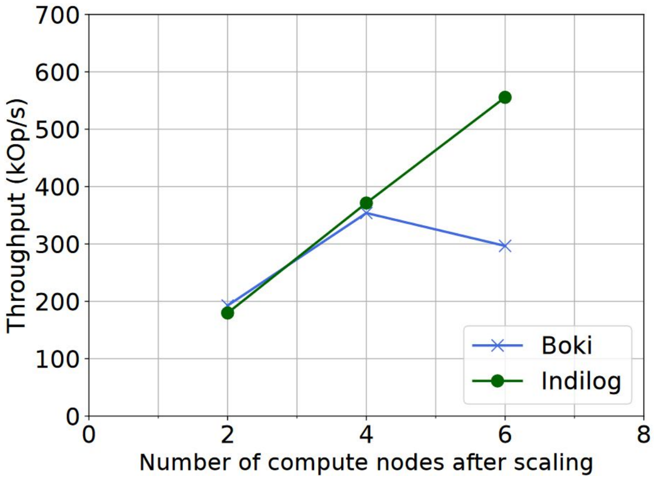
Workload: 50/50 Append/Read + 20% Local Index Hit Ratio in IndiLog



A low index hit ratio in IndiLog lowers the overall throughput

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

Workload: 5/95 Append/Read + IndiLog: 87% Local Index Hit Ratio



IndiLog's throughput scales with the number of nodes
Boki's node with the complete index gets under heavy contention

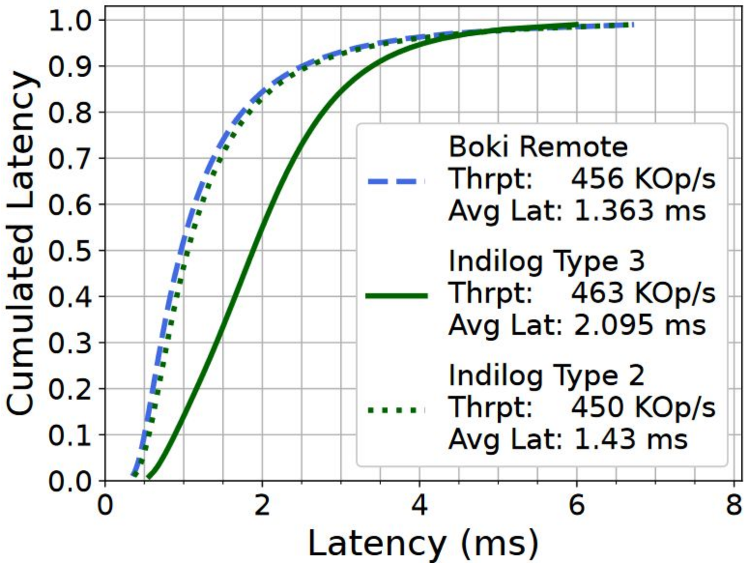
Read latencies of the index tier

IndiLog

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

Boki

- <u>2 more compute nodes</u> maintain complete indexes but do not run functions
- 4 compute nodes with functions do remote index lookups only



IndiLog's sharded index tier comparable to remote complete indexes

Real application

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

- Workload: functions of 10k users doing CRUD operations on key-value objects for 30 seconds

System	Reads	Local Index Hit Ratio	Throughput [Op/s]
IndiLog - default ~ 20 MB local index	2.072M	0.93	8700
IndiLog - small ~ 0.2 MB local index	1.873M	0.47	8430

Boki throughput:
8950 Op/s

IndiLog's performance comparable with Boki
Even with a small index IndiLog captures almost 50% of lookups locally

Conclusion

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 computer tier