Effective Workflow Auto-Scaling in Streaming System.

By Zikun Wang, advised by John Liagouris
Department of Computer Science

Abstract

State-of-the-art stream processors partition data across nodes in the cluster to enable data-parallel processing. To keep up with workload changes, we often need to scale out the system by adding more nodes. However, adding (or removing) nodes requires re-partitioning the system state, which in turn requires efficient data movement between machines at runtime. Existing state migration strategies either incur large overhead on steady state or require system downtime. In this project, we propose a streaming system capable of auto-scaling with zero downtime and minimal overhead on steady state by implementing a novel state migration design that decouples compute and storage. In our design, when a new machine is added to the system while it is running, the system state is lazily fetched only when the machine requires it, effectively treating the old workers as external storage. We implemented most of the system runtime (e.g., metric collection, watermark propagation) and the baseline migration strategies over the summer. Our system performs well compared to streaming systems like Flink[1] and Storm[2]. In the future, we aim to complete the implementation and optimization of our hybrid state migration technique and compare it with state-of-the-art approaches.

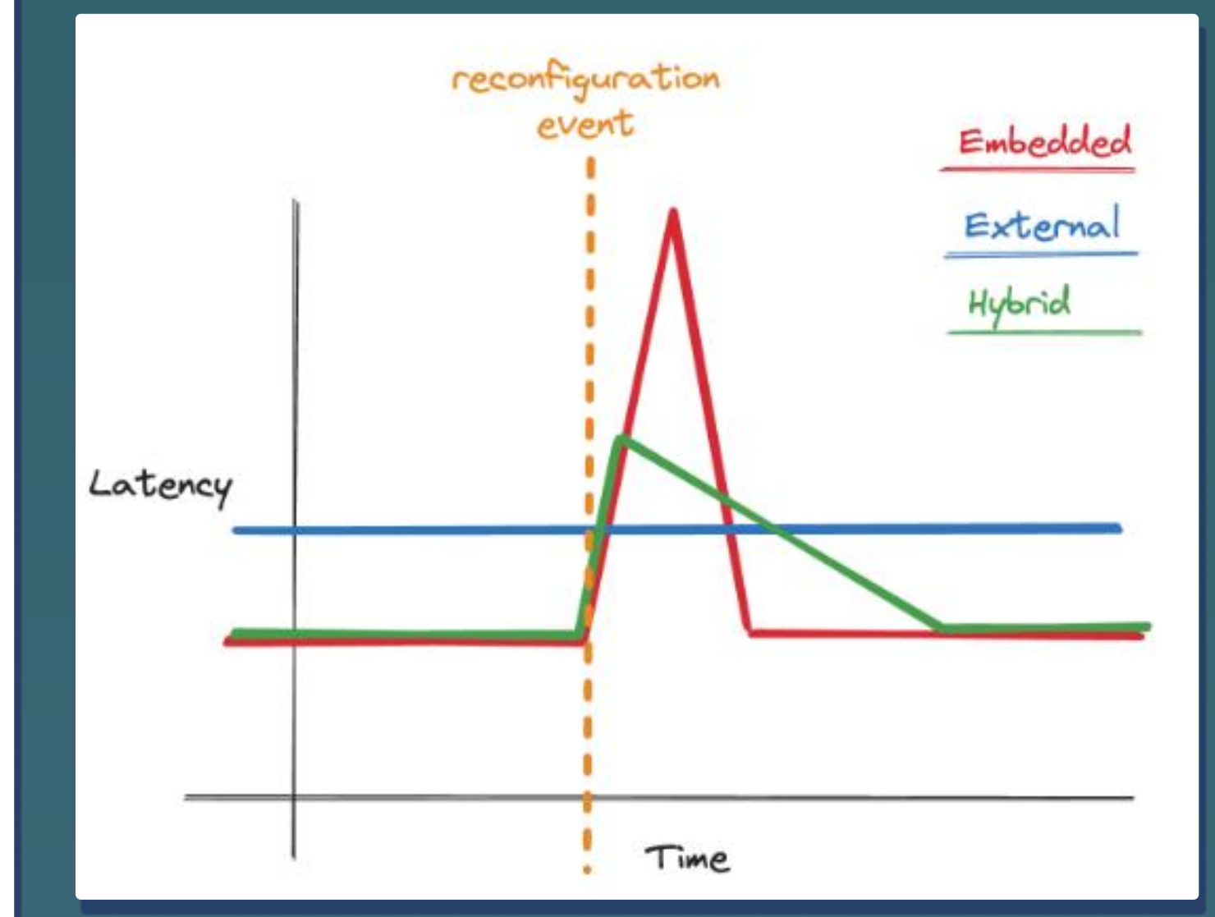


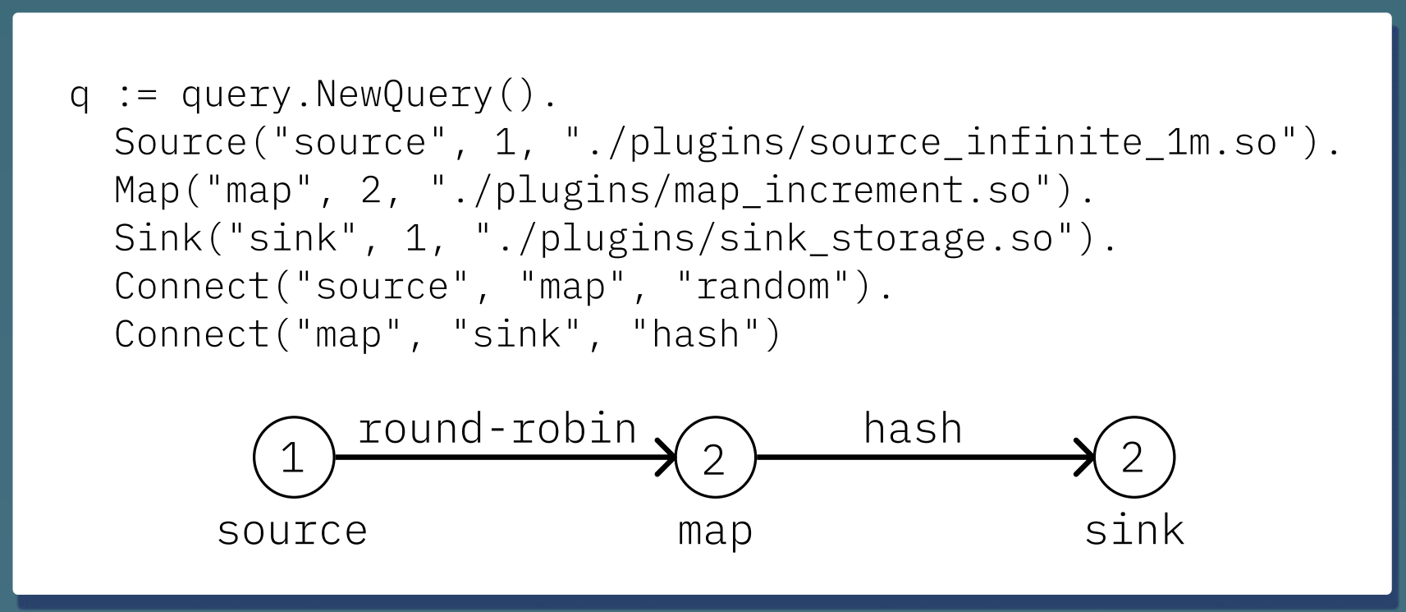
Figure 1. Using hybrid state migration strategy to reduce latency spike during system re-configurations.

Runtime Architecture

We developed a distributed streaming processor runtime in Go from the ground up, with gRPC library for our networking.

The processing model is similar to Flink. For each cluster, we have one coordinator and multiple workers (possibly run on different machines). The user will create a dataflow query locally using our client API, and then submit the query to the coordinator. The coordinator will deploy the abstract query plan to the registered workers.

For each worker, when getting assigned to a stateful job (like counter), the user has the ability to choose between local storage (in-memory or PebbleDB[4]) or remote storage (connecting to a TiKV[5] cluster).



```
q := query.NewQuery().
  Source("source", 1, "./plugins/source_infinite_1m.so").
  Map("map", 2, "./plugins/map_increment.so").
  Sink("sink", 1, "./plugins/sink_storage.so").
  Connect("source", "map", "random").
  Connect("map", "sink", "hash")
```

Figure 2. Generating dataflow typology through our user query API.

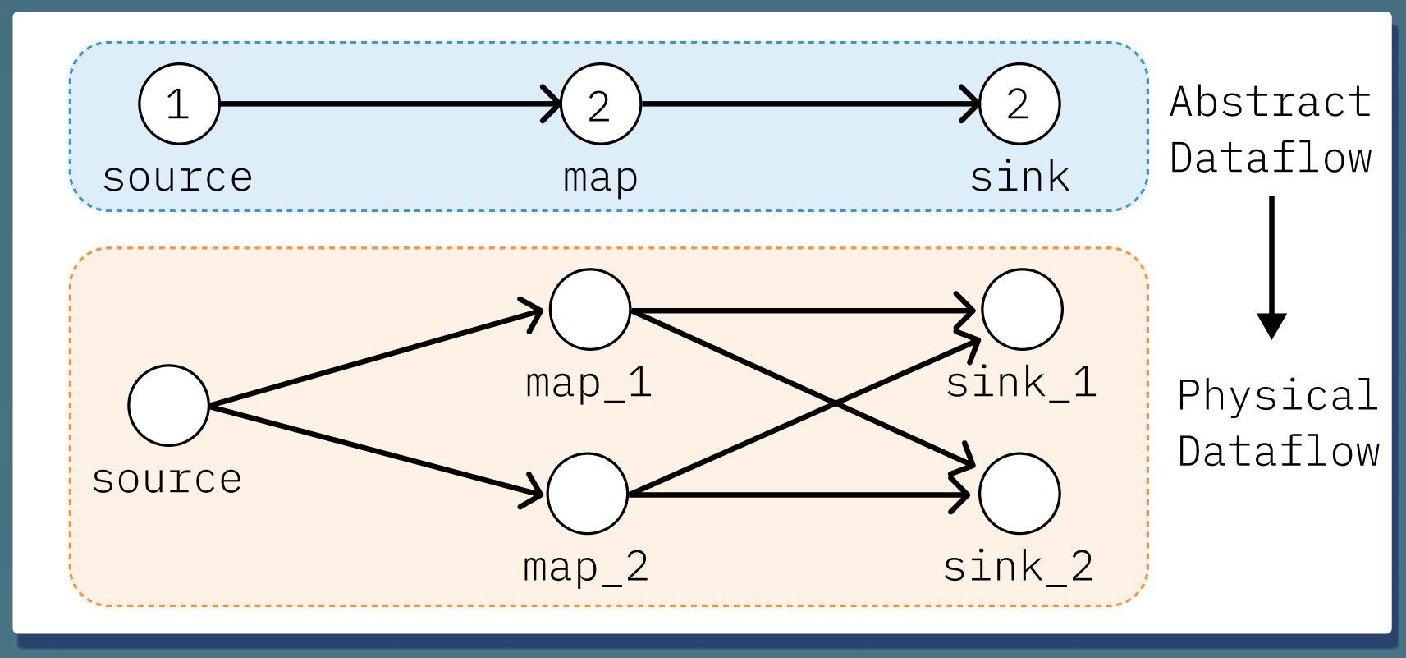


Figure 3. Mapping abstract dataflow to physical plans that we can deploy to workers.

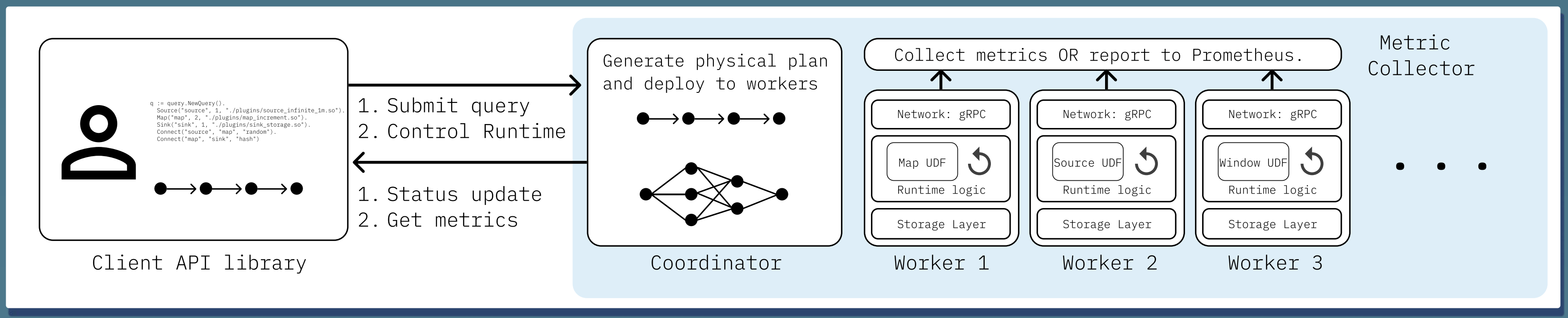
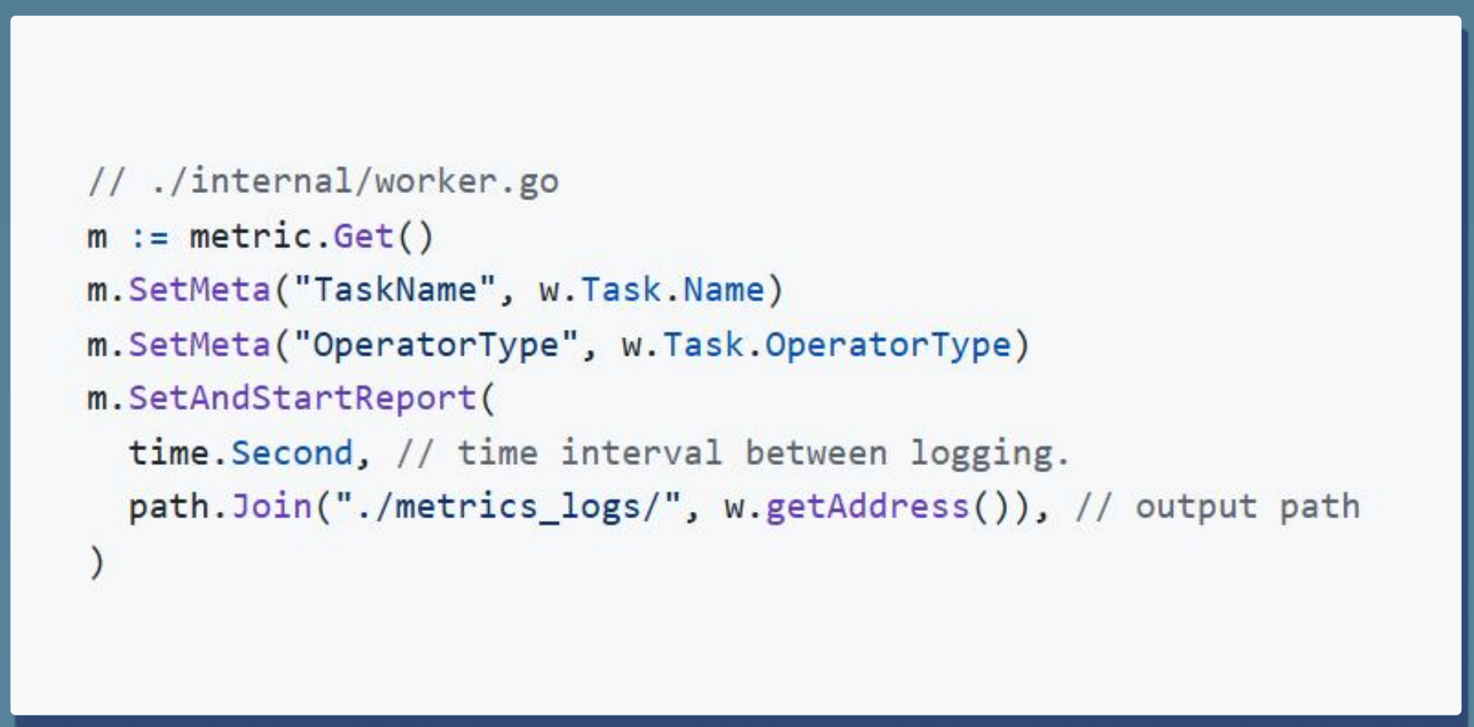


Figure 4. Runtime overview.

Benchmarks and Metrics Collection

To understand the performance of our runtime, we ran benchmarks on cloud machines on platforms such as Chameleon[6] and CloudLab[7], then record the average throughput and tail latency. We developed a metric collection package in Go to achieve this, as well as support analyzing metrics in Prometheus[8] in real-time for debugging purposes.



```
// ./internal/worker.go
m := metric.Get()
m.SetMeta("TaskName", w.Task.Name)
m.SetMeta("OperatorType", w.Task.OperatorType)
m.SetAndStartReport(
  time.Second, // time interval between logging.
  path.Join("./metrics_logs/", w.getAddress()), // output path
)
```

Figure 5. Metrics collection package API initialization.

Figure 6. Analyze throughput and latency information through Prometheus in real-time.

Future Work

We are keep developing our runtime, adding more features and improving its performance, especially in areas such as better network management and buffer management. In the same time, we are working on improving the lazy fetch approach, solving the performance issues we observed in our initial demo.

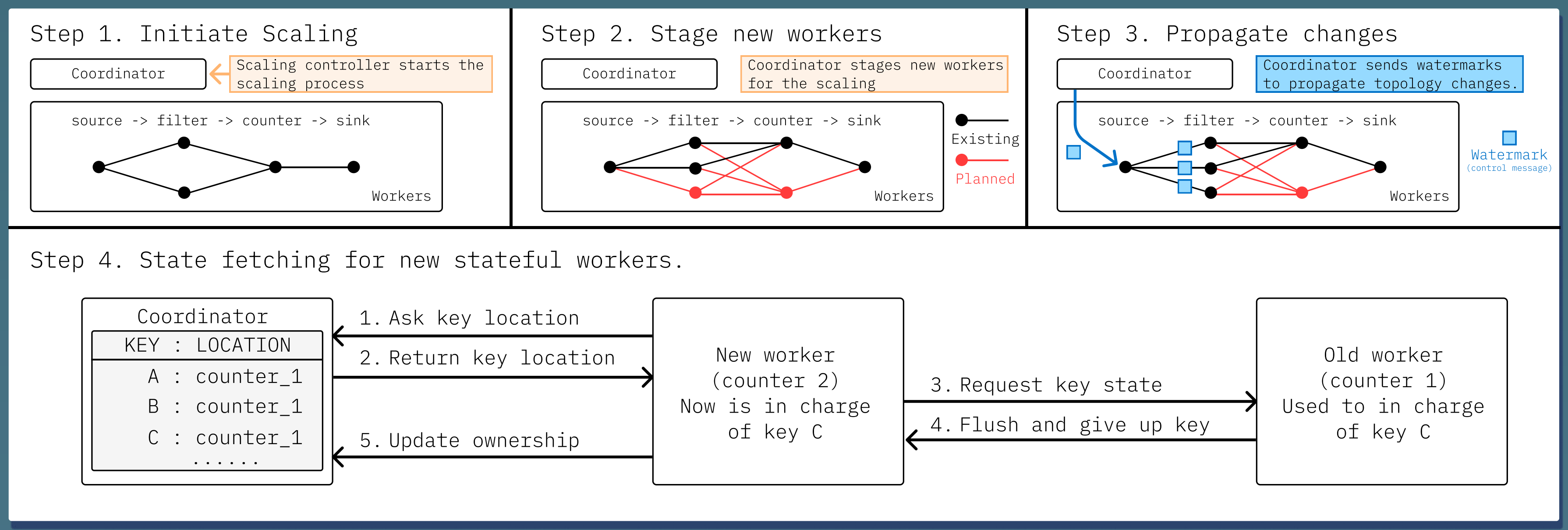
Special Thanks

This project was made possible thanks to my professors, John Liagouris and Vasia Kalavri. Their guidance has been crucial in helping me navigate the complexities of distributed systems.

I am also deeply grateful to PhD students Yuanli Wang, Lei Huang, and the other wonderful members of the CASP Lab. They provided immense help and support during my most challenging times.

Lazy State Migration Strategy

We designed a lazy state migration during scaling. The key idea is the new workers will effectively treat the old workers as external storage. After fetching the corresponding key value pairs from the old workers, the new workers will also gain the ownership of the data, completing the state migration.


Step 1. Initiate Scaling
Coordinator
Scaling controller starts the scaling process
source -> filter -> counter -> sink
Workers
Step 2. Stage new workers
Coordinator
Coordinator stages new workers for the scaling
source -> filter -> counter -> sink
Workers
Existing
Planned
Step 3. Propagate changes
Coordinator
Coordinator sends watermarks to propagate topology changes.
source -> filter -> counter -> sink
Workers
Watermark (control message)
Step 4. State fetching for new stateful workers.
Coordinator
KEY : LOCATION
A : counter_1
B : counter_1
C : counter_1
......
1. Ask key location
2. Return key location
5. Update ownership
New worker (counter 2) Now is in charge of key C
3. Request key state
4. Flush and give up key
Old worker (counter 1) Used to in charge of key C
Figure 7. What happens during a scale up using the lazy state migration strategy.

Results

To verify the scaling behavior and performance of our base runtime, we ran the following experiment on CloudLab machines. Each machine has two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz (40 threads in total), 192GB DDR4 memory, and 480GB SATA SSD. For each experiment, we use a simple source→map→sink query (source→counter→sink for stateful experiments). We fixed the number of sources and sinks, and varies the number of maps/counters. In the end, we record the throughputs and tail latencies.

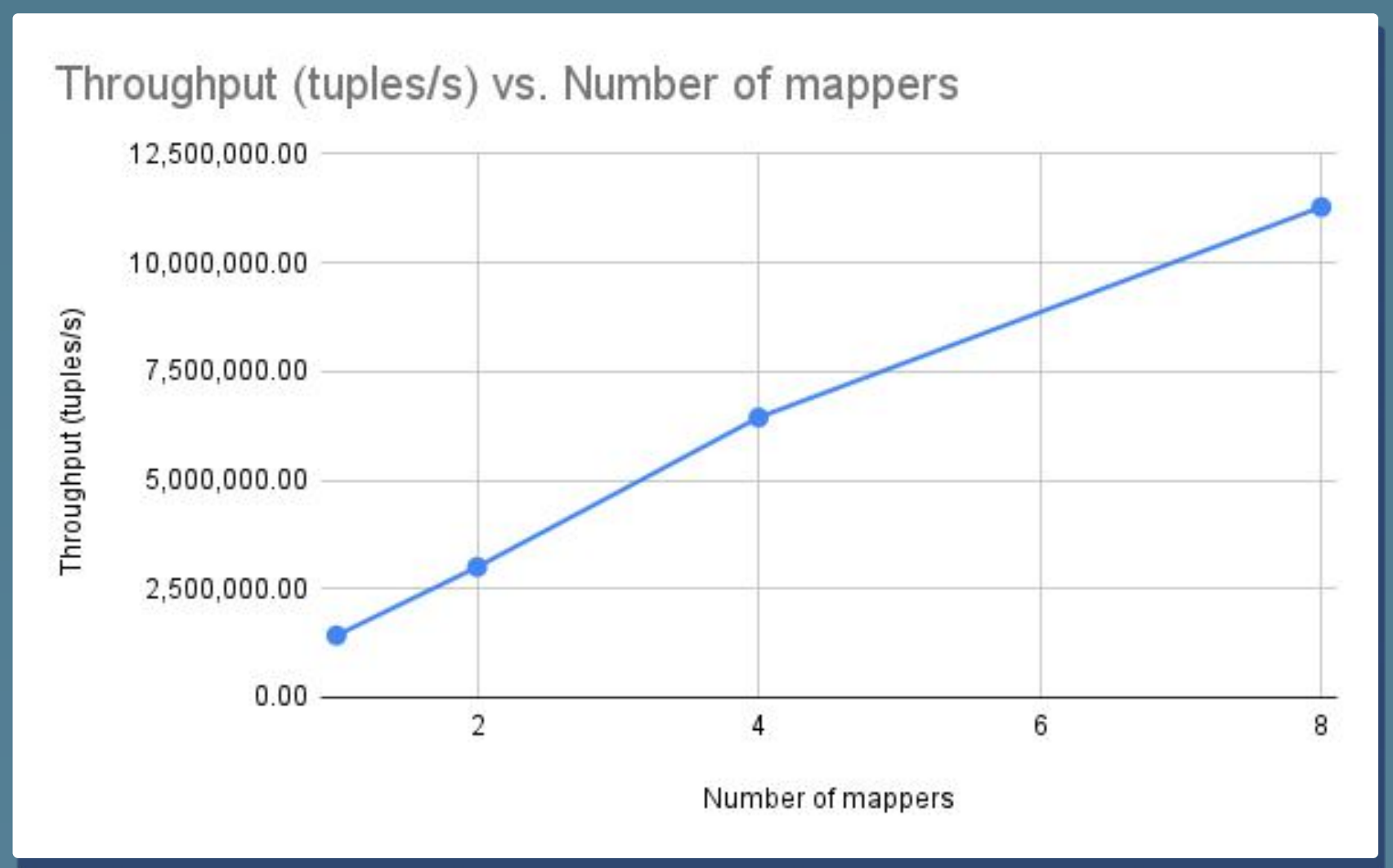


Figure 8. Stateless experiment on a single node machine.

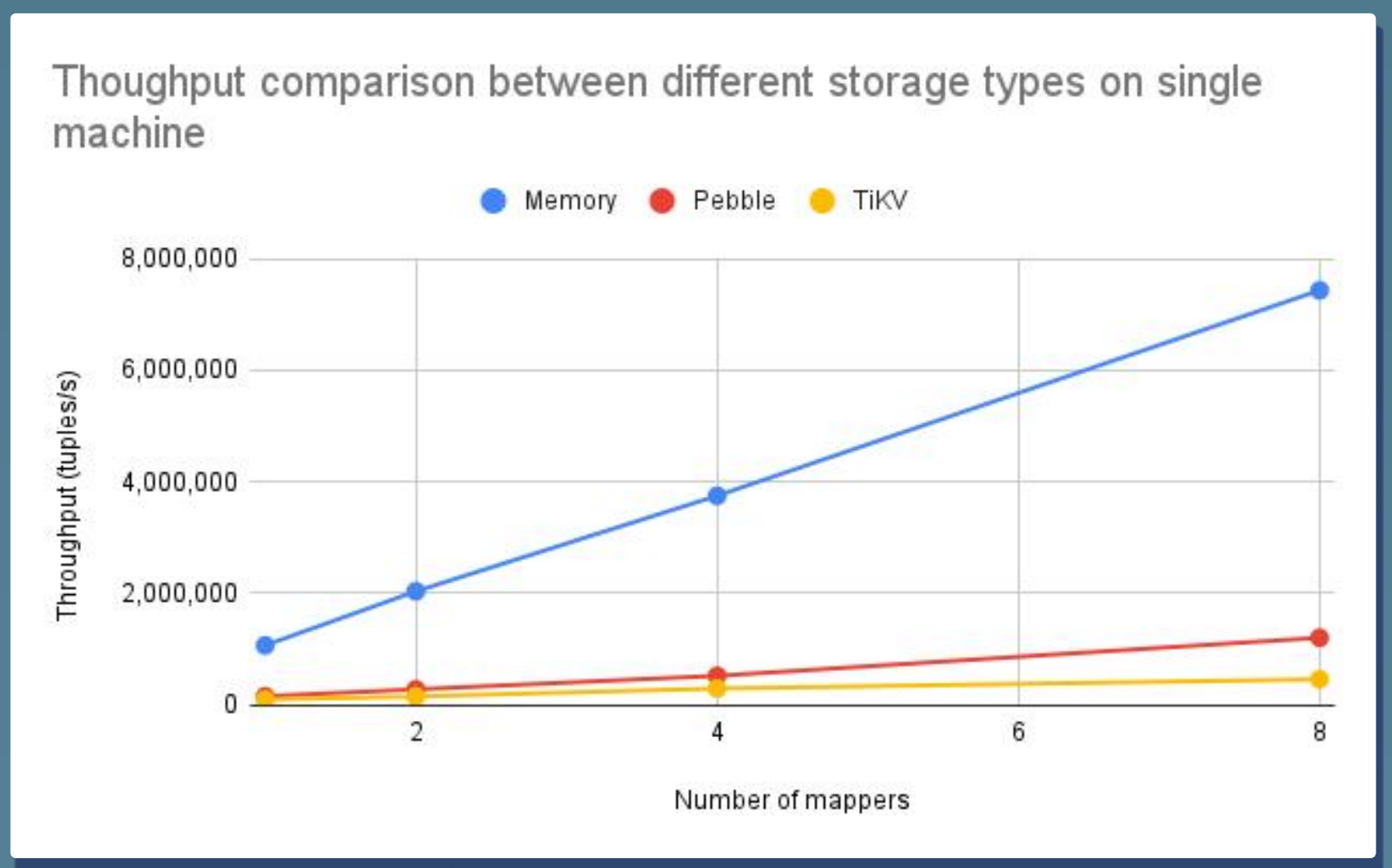


Figure 9. Stateful experiment on a single machine.

For experiment running on multiple machines, we placed all the sources on one node, all the sinks on one node, and split the counters on two nodes. (Each worker uses two threads)

We achieved high throughput with our runtime with good scaling behavior when increasing the available physical resources.

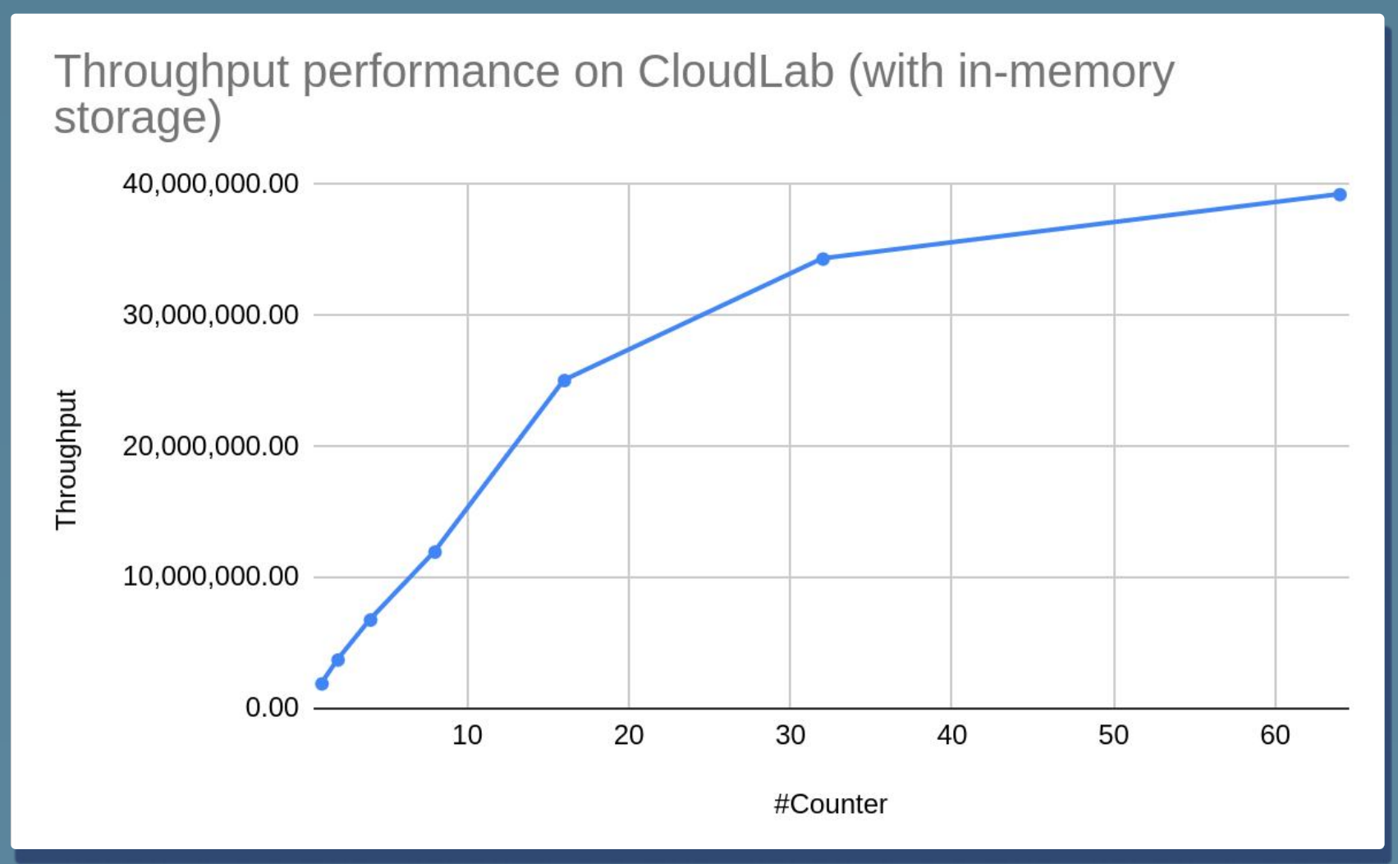


Figure 10. Stateful experiment on multiple machines.

References

[1] Apache Flink: State Backend. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/ops/state/state_backends/. Last access: Sep 2024.
[2] Apache Storm. https://storm.apache.org/. Last access: Sep 2024.
[3] gRPC. https://grpc.io/. Last access: Sep 2024.
[4] PebbleDB on Github. https://github.com/cockroachdb/pebble. Last access: Sep 2024.
[5] TiKV. https://tikv.org/. Last access: Sep 2024.
[6] Chaemelon Cloud: https://www.chameleoncloud.org/. Last access: Sep 24.
[7] CloudLab. https://www.cloudlab.us/. Last access: Sep 2024.
[8] Prometheus. https://prometheus.io/. Last access: Sep 2024.