Carnegie Mellon University

Parpulse: I/O Service for Modern OLAP Database System

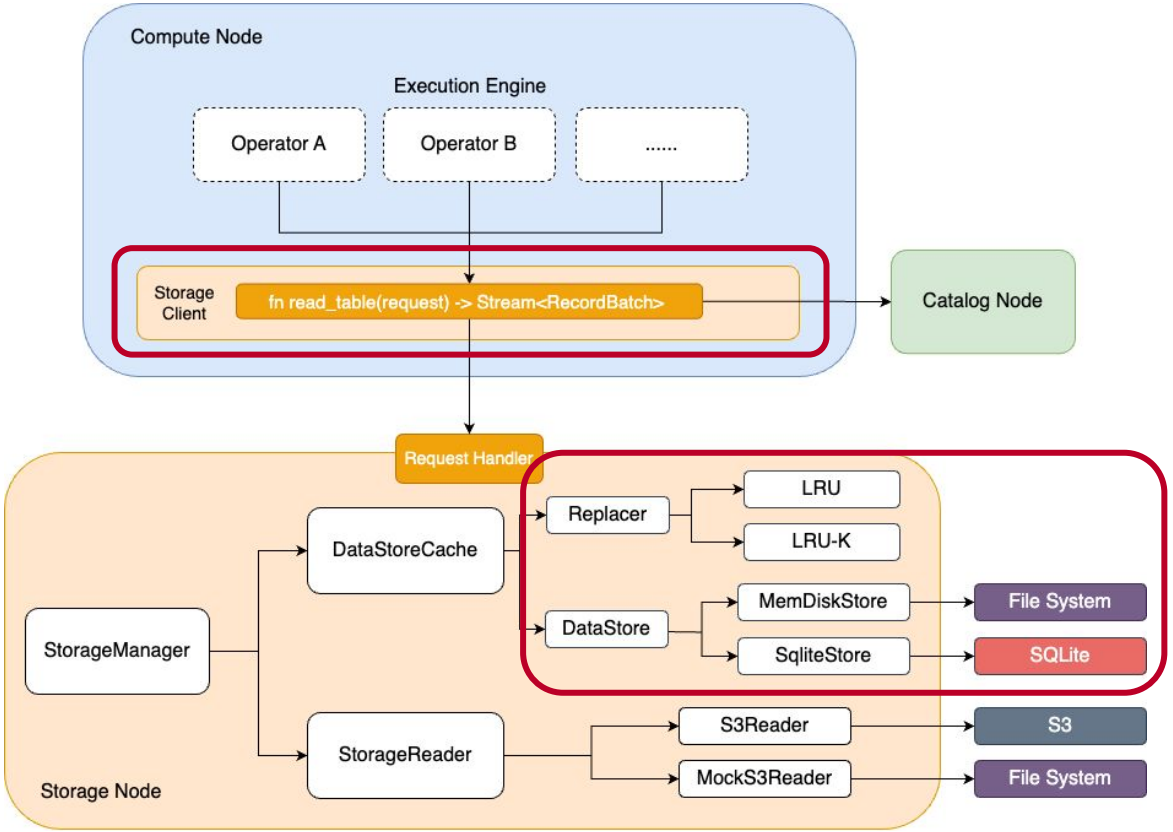
Yuanxin Cao, Lan Lou, Kunle Li

CMU-DB

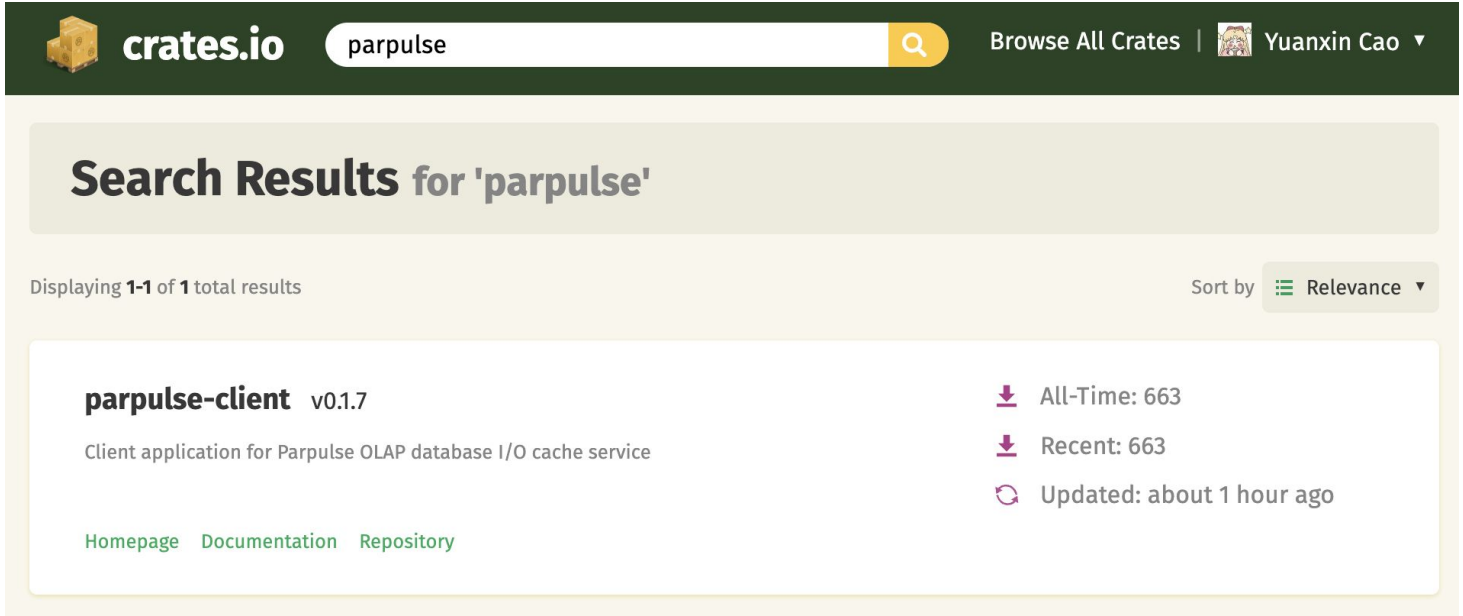
Current Status

- 🟠 **75% Goal** – Build a functional I/O Service
  - ✅ Support reading data from the underlying storage (e.g. S3)
  - ✅ Implement a local cache with different cache policies on the Storage Node for fast data retrieval
  - ✅ Send requests from storage client to storage node
- 🟢 **100% Goal** – Optimization
  - ✅ Add memory cache for small data
  - ✅ Add more parallelism with async and fine-grained lock
    - Handle 2 requests with the same key at the same time efficiently
    - Pull next data and push current data at the same time
  - ✅ Set up an E2E automatic benchmark pipeline
- 🔥**125% Goal** – More optimizations……
  - ✅ Optimize the storage reader for high-performance reading (I/O request merging)
  - 🔲 Develop extra features such as prefetching, kernel bypassing for data reading

Architecture



Storage Client



Storage Client

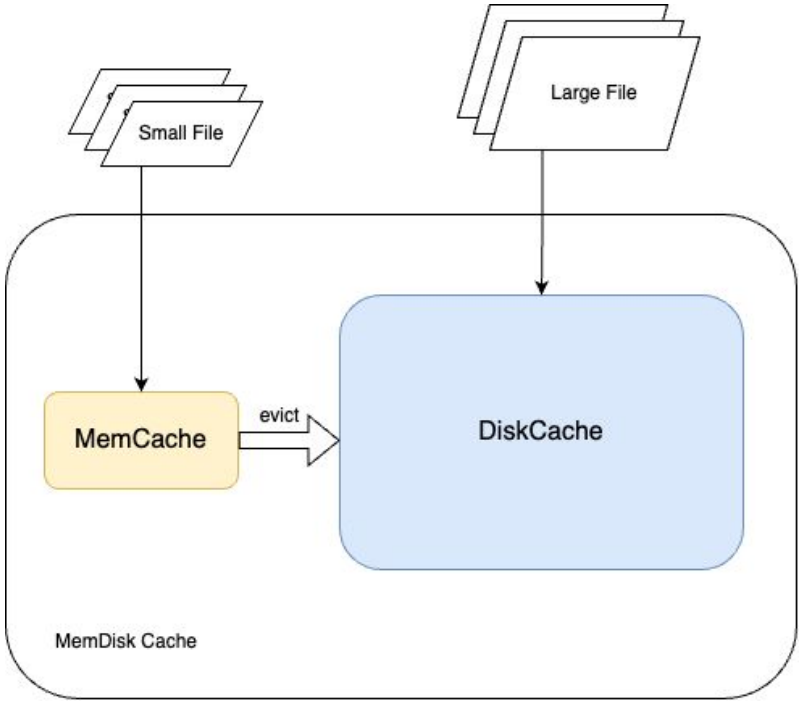
ParpulseStorageClientImpl (I/O Service Team 1)

IstziioStorageClientImpl (I/O Service Team 2)

Same basic logics:

- **Send request to server and get data back (Parquet)**
- **Decode Parquert -> Arrow**
- **Stateless!**

Mem-Disk Data Store Cache



Fine-grained Lock

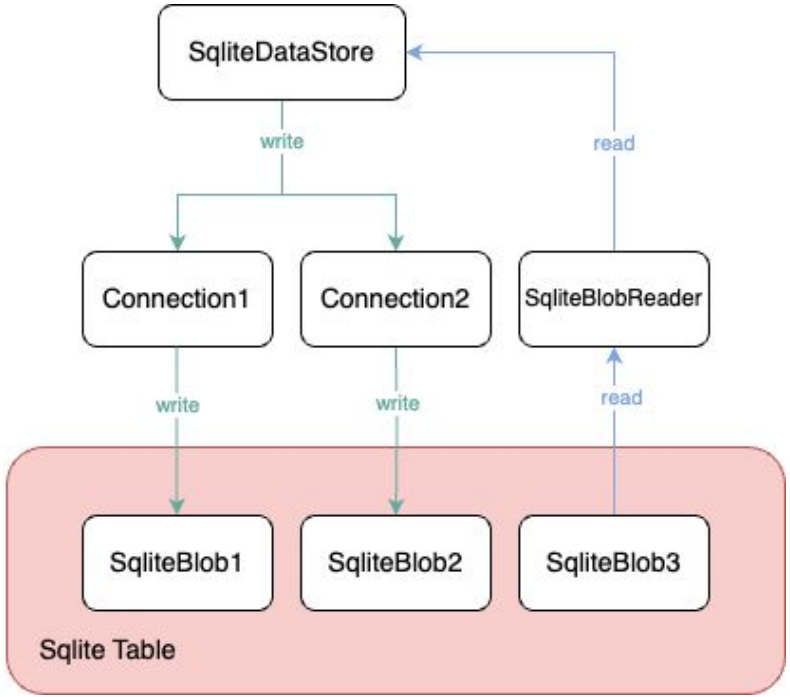
Serve Parallel Requests!

Sqlite Data Store Cache

1 Parquet ⇔ 1 Sqlite Blob

parpulse_cache	
pk	row_id
	content (blob)

Cache Key: row_id



Benchmark

- **Dataset**

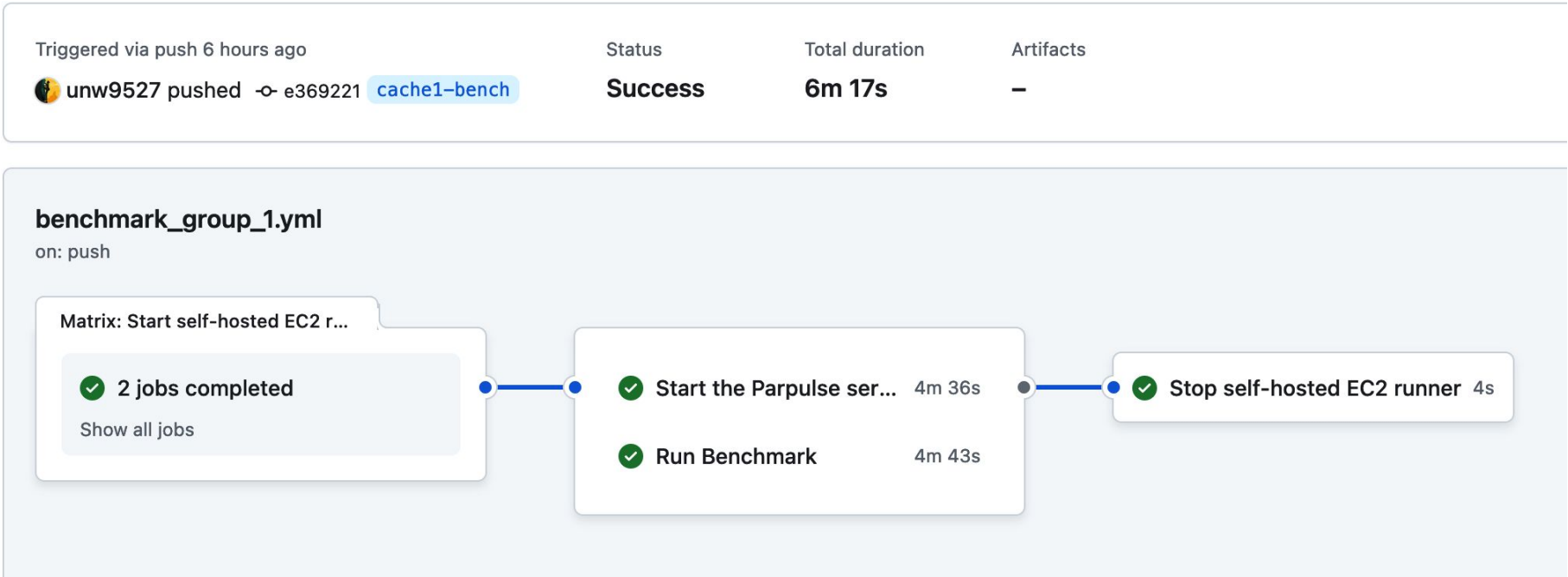
```
---------- First 5 rows of the parquet file ----------
      col 1     col 2     col 3     col 4     col 5     col 6     col 7  ...
0  0.191954  0.481544  0.470787  0.779391  0.218772  0.994886  0.119190  ...
1  0.841532  0.687789  0.691565  0.930984  0.852517  0.107169  0.909794  ...
2  0.606424  0.839445  0.475433  0.842104  0.789378  0.456189  0.564459  ...
3  0.474810  0.274558  0.159864  0.478374  0.891250  0.247201  0.624717  ...
4  0.933724  0.145045  0.506208  0.020810  0.916214  0.142837  0.955672  ...
```

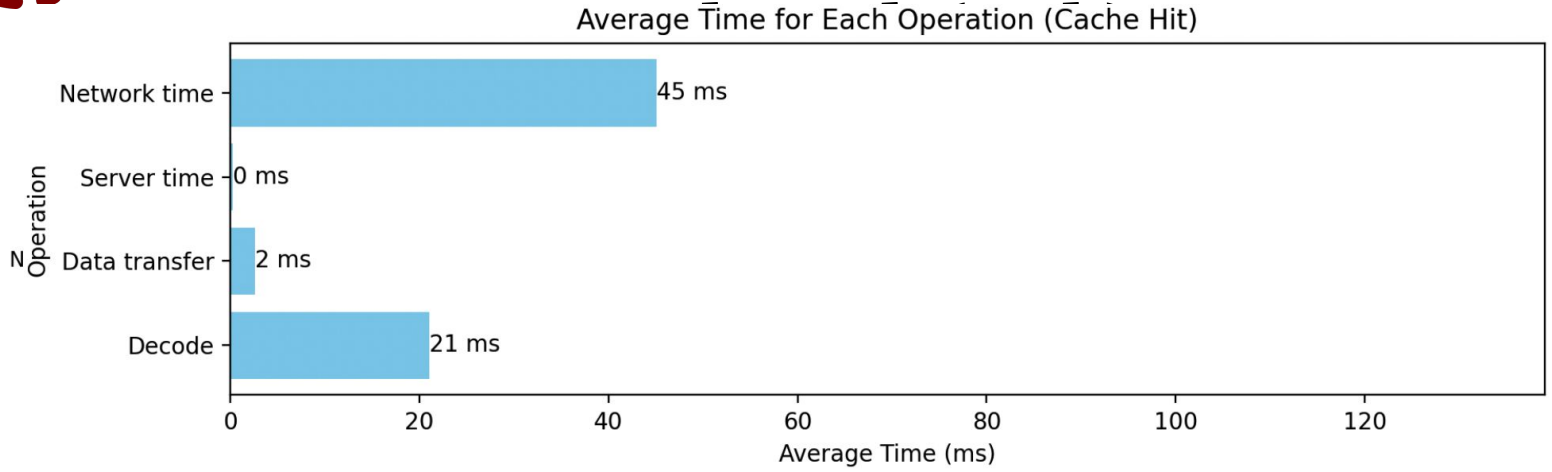
  - | Arrival Time | File Index |
- **Metric**
  - E2E time for **client** and **server**
- **Access Pattern**
  - Zipfian
- **Machine**
  - AWS EC2 (ubuntu 22.04, C5.xlarge → 4vCPU, 8GB memor

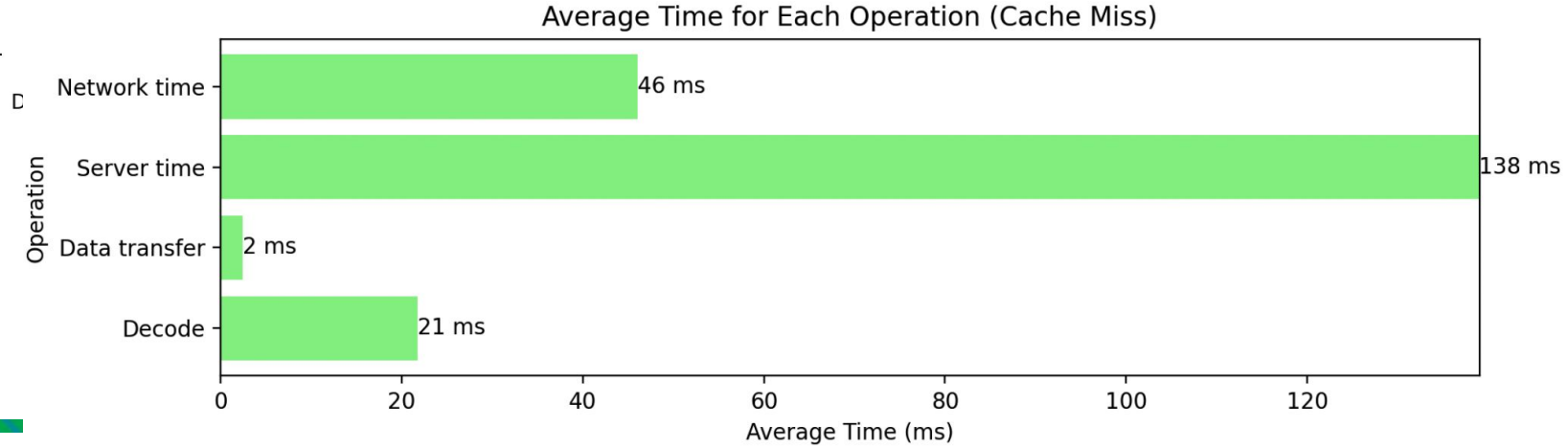
S =

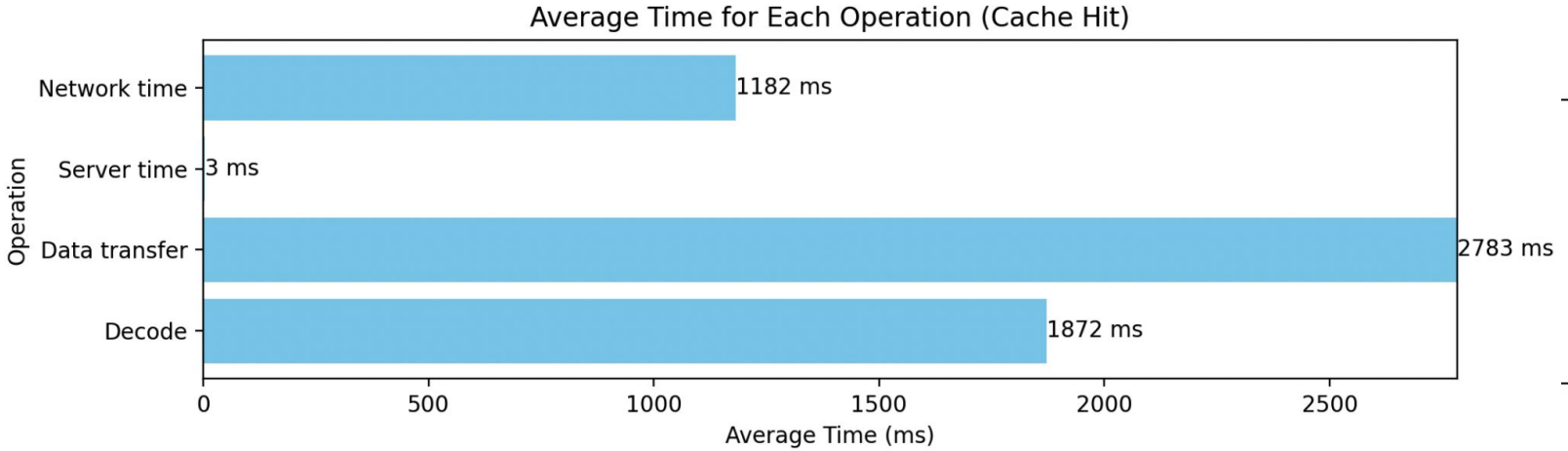
1	timestamp	file_index
2	0	11
3	8	10
4	365	10
5	688	10
6	1123	10
7	1203	10
8	1213	11

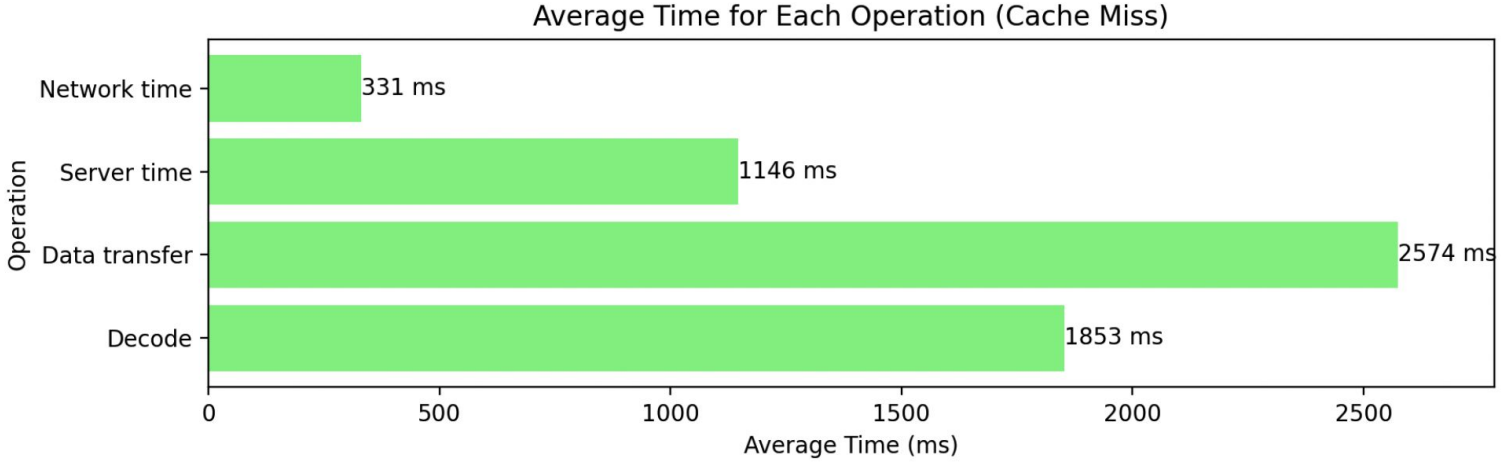
Whole process is triggered in GitHub Action!

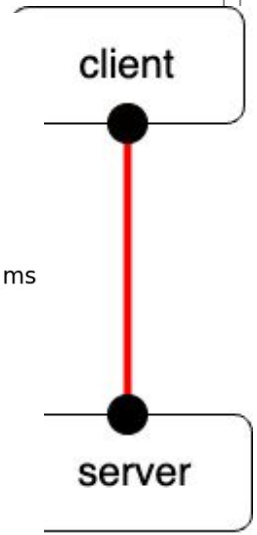


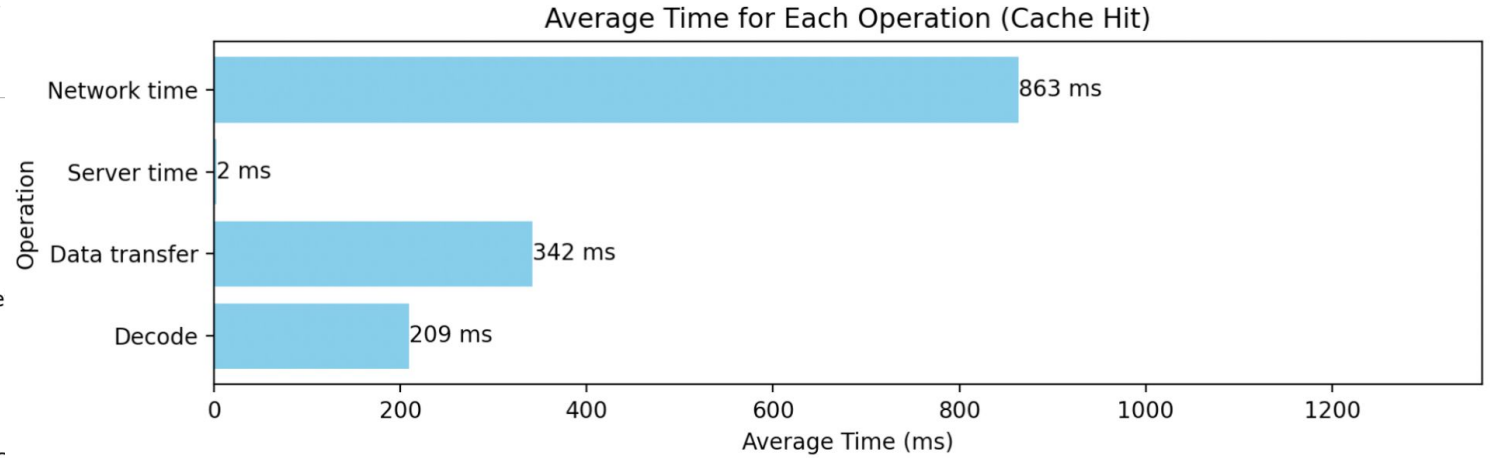

Average Time for Each Operation (Cache Hit)
Network time
Server time
Data transfer
Decode
45 ms
0 ms
2 ms
21 ms
Operation
0
20
40
60
80
100
120
Average Time (ms)

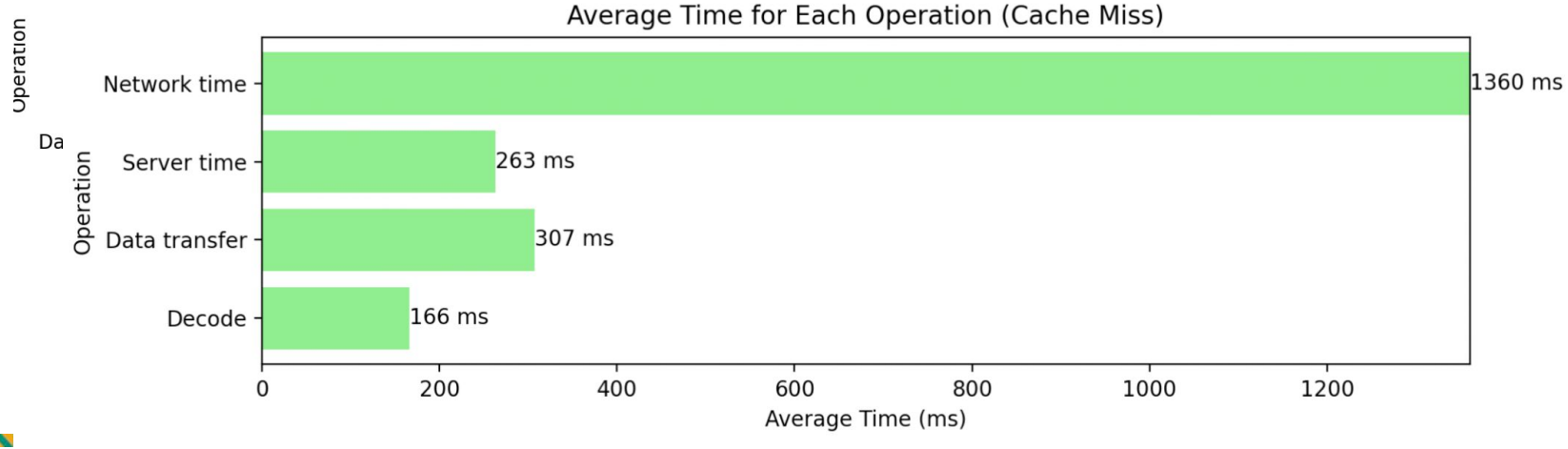

Average Time for Each Operation (Cache Miss)
Network time
Server time
Data transfer
Decode
46 ms
138 ms
2 ms
21 ms
Operation
0
20
40
60
80
100
120
Average Time (ms)

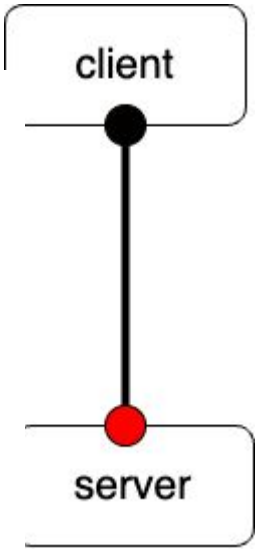

Average Time for Each Operation (Cache Hit)
Network time
Server time
Data transfer
Decode
Operation
1182 ms
3 ms
2783 ms
1872 ms
0
500
1000
1500
2000
2500
Average Time (ms)


Average Time for Each Operation (Cache Miss)
Network time
Server time
Data transfer
Decode
Operation
331 ms
1146 ms
2574 ms
1853 ms
0
500
1000
1500
2000
2500
Average Time (ms)


client
server


Average Time for Each Operation (Cache Hit)
Operation
Network time
Server time
Data transfer
Decode
863 ms
2 ms
342 ms
209 ms
0
200
400
600
800
1000
1200
Average Time (ms)


Average Time for Each Operation (Cache Miss)
Operation
Network time
Server time
Data transfer
Decode
1360 ms
263 ms
307 ms
166 ms
0
200
400
600
800
1000
1200
Average Time (ms)


client
server

Discoveries

- Server time (Server E2E time, including polling data from S3, storing it into local cache, return receiver channel) decreases significantly when cache hits
- **Bottleneck**
  - Big data: data transfer
  - Small data: HTTP setup
- When there are too many requests at the same time
  - Data transfer time will be long → B/W not enough
  - If TOO MANY → *Wait time will be long* → server cannot handle, requests stuck in client

Benchmark

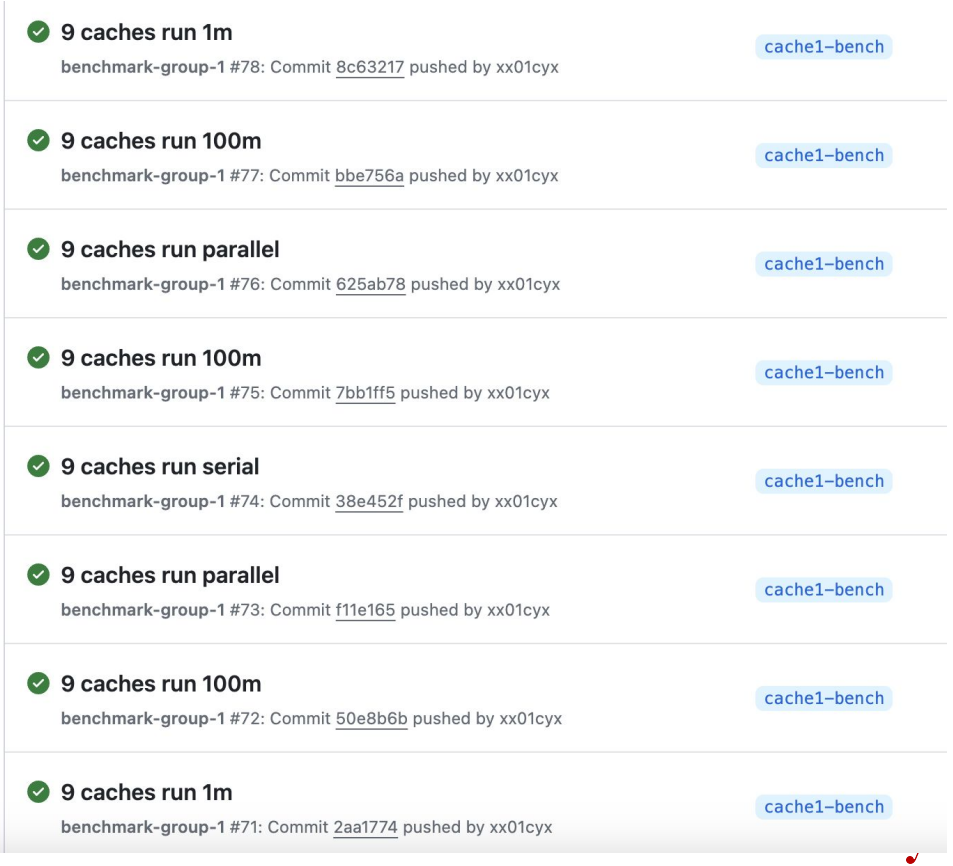
Trace	Average time (ms)	Network Time (ms)	Server Time (ms)	Data Transfer Time (ms)	Decode Time (ms)	Result Link
1m	105.569877	45	34	2	21	link
100m	5913.843632	714	631	2668	1862	link
Parallel	1958.889723	1168	162	320	182	link*
Serial	765.985548	40	244	63	409	link

100m trace for multiple runs: 5.924315929s, 6.683559418s, 6.004198074s **(6-7s)**

* From benchmark result, it is parallel test, but we wrongly set the commit message : (

Benchmark

- Where we get our result: https://github.com/cmu-db/15721-s24-cache-benchmark/actions/workflows/benchmark_group_1.yml



Single 100M Request

(screenshot from **serial** trace)

timestamp	file_index
10000	10

All requests are cache miss!
(timestamp unit: ms, arrive time for each request)

Server Time: 1.663519s

Parallel Multiple 100M Request

	timestamp	file_index
2	0	11
3	250	11
4	255	12
5	265	13
6	278	14
7	500	14
8	510	15

Avg Server Time: 2.053836s
Fanout cache num = 1 link

Avg Server Time: 1.795651s
Fanout cache num = 3 link

Memory Cache

- > 10 mb (large file) -> disk
- <= 10 mb (small file) -> memory
- don't need to send extra S3 request to get size
- eviction -> write to disk cache

Disk Cache

Basic Server Logic

- First get from cache -> hit?
- If hit, read & return
- If miss, ...
  - put data to cache
    - poll from s3
    - write data to cache
  - get data from cache again
  - return receiver channel

Server Time

Fine-grained Lock + Unlock Disk Manager

write **read** **evict**

- **Write & Write**: Complete Status for keys + notify waiters
  - "status_of_key": hashmap with completed/uncompleted status for each key
  - when requests come into put_data, see uncompleted, sleep to be notified
  - see nothing, insert incompleted, put_data, then notify all waiters

- **Get & Put Atomicity**: Get -> Put (but data in cache) -> Get (but data evicted)
  - status_of_keys also record all the keys in mem_replacer + disk_replacer
  - when requests come into put_data, see completed, pin data, directly return

Fine-grained Lock + Unlock Disk Manager

write **read** **evict**

- **Evict & Read**: pin & unpin data in replacer
  - Pin data when using (transfer to network, between put & get)
- **Write & Evict**: correctly update “status_of_keys”
  - Mem evict: lock “status_of_keys”
  - Disk evict: remove from “status_of_keys”
- **Write & Read**: First write to cache, then write to replacer
  - no need to lock replacer when writing data to mem/disk
  - if putting to replacer fails, then clean the mem/disk
  - “optimistic put”

Other Optimizations

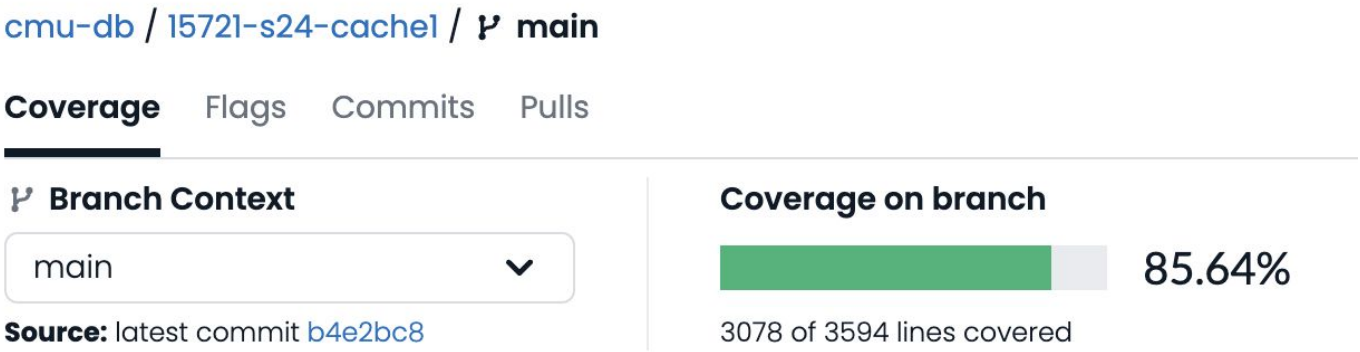
- **Fanout Cache (Benchmark is set to 9)**

https://grantjenks.com/docs/diskcache/tutorial.html#fanoutcache

```
index = self._hash(key) % self._count
cache = self._caches[index]
try:
    # lock `cache`
    return cache.add(key, value, expire, read, tag, retry)
```

- **Write current data to disk and poll next data from S3 at the same time**

Code Coverage Report



Future work

- Predicate pushdown to storage node
- Kernel bypass when reading data (io_uring)
- Eliminate disk I/O on Storage Client
- Network improvement…

Carnegie Mellon University

Thank You!