Practical Course: Cloud-Based Data Processing

Michalis Georgoulakis

Chair for Database Systems
School of Computation, Information and Technology
Technical University of Munich

19.05.2026

Agenda

Lab 3 is over

- Broadcast vs Shuffle Hash Join

Today

- Lab 3 Discussion
- Cloud Storage
- Io_uring

Schedule Update

	Content	Lab Start	Lab End	Oral Discussions
19.05	Cloud Storage	Lab 4	Lab 3	
26.05	**No Session** (Whit Tuesday)	Lab 5		
02.06	**No Session** (Conference Trip)		Lab 4	
09.06	Cloud Compute & Networking	Lab 6	Lab 5	Lab 3, Lab 4
16.06	Cloud Benchmarking	Project	Lab 6	Lab 5
23.06	Topic Selection			Lab 6

Try not to miss the June sessions

Cloud Storage: Where does data live?

CLOSED SYSTEM

Postgres, MySQL, classical OLAP

- The DBMS owns the data.
- Data lives in its own pages, on its own disk, in its own format.
- The DB controls layout, sort order, compression, statistics.

OPEN SYSTEM

Distributed data systems

- CSV, Parquet, JSON on S3 / Azure / GCS

The new question: *given an open system, how fast can you actually read the data off cloud storage?*

What one can get with a single TCP connection

THE BANDWIDTH-DELAY PRODUCT

What a single TCP connection from your VM to S3 in the same region actually delivers

4 MB	÷ 10 ms	× 8	= 3.2 Gbps
TCP window (typical)	*RTT to S3*	*bits per byte*	*per connection, ceiling*

One thread, one connection, one cap

THE BANDWIDTH-DELAY PRODUCT

What a single TCP connection from your VM to S3 in the same region actually delivers

4 MB	÷ 10 ms	× 8	= 3.2 Gbps
TCP window (typical)	*RTT to S3*	*bits per byte*	*per connection, ceiling*

ONE CONNECTION

Caps at ~400 MB/s

And that's a generous ceiling. In practice TLS, slow-start, and CWND recovery after packet loss bring real-world per-connection throughput closer to 100–200 MB/s.

One thread, one connection, one cap

THE BANDWIDTH-DELAY PRODUCT

What a single TCP connection from your VM to S3 in the same region actually delivers

4 MB	÷ 10 ms	× 8	= 3.2 Gbps
TCP window (typical)	*RTT to S3*	*bits per byte*	*per connection, ceiling*

ONE CONNECTION

Caps at ~400 MB/s

And that's a generous ceiling. In practice TLS, slow-start, and CWND recovery after packet loss bring real-world per-connection throughput closer to 100–200 MB/s.

TO FILL A 10 Gbps NIC

You need many connections

10 Gbps ÷ 200 MB/s = ~50 concurrent connections, minimum. And every connection costs a TLS handshake, file descriptor, and kernel memory.

One thread, one connection, one cap

THE BANDWIDTH-DELAY PRODUCT

What a single TCP connection from your VM to S3 in the same region actually delivers

4 MB	÷ 10 ms	× 8	= 3.2 Gbps
TCP window (typical)	*RTT to S3*	*bits per byte*	*per connection, ceiling*

ONE CONNECTION

Caps at ~400 MB/s

And that's a generous ceiling. In practice TLS, slow-start, and CWND recovery after packet loss bring real-world per-connection throughput closer to 100–200 MB/s.

TO FILL A 10 Gbps NIC

You need many connections

10 Gbps ÷ 200 MB/s = ~50 concurrent connections, minimum. And every connection costs a TLS handshake, file descriptor, and kernel memory.

Parallelism is the only way out. The question is how to make it cheap.

Four options

OPTION 1

Many blocking threads

One thread per request, each blocks on read().

- Simple but expensive
- Context-switch overhead, kernel stacks, lock contention.

OPTION 2

epoll + non-blocking

One thread monitors many fds, issues reads when ready.

- Better performance
- Every operation is still two syscalls: poll for readiness, then actually read.

OPTION 3

POSIX AIO

True async I/O — submit operation, get notified when done.

- Designed for exactly this use case.
- The Linux implementation is widely considered broken, falls back to threads, limited file-type support.

OPTION 4

io_uring

Submit a batch of operations through a shared ring buffer, the kernel works in the background, completions appear in a second ring.

- Per-operation syscall cost → effectively zero.

Blocking Threads on Read

The simplest approach.

1969 POSIX threads + blocking I/O. “The recipe everyone tries first.” *since forever*

1 thread

per request

Each thread owns its read(). It blocks until bytes arrive. No async machinery anywhere.

~8 KB

kernel stack per thread

Plus a 1–8 MB virtual userspace stack.

1000s/sec

context switches under load

Every read() + every block + every wakeup is a kernel trip.

Blocking Threads on Read

The simplest approach.

1969 POSIX threads + blocking I/O. “The recipe everyone tries first.” *since forever*

1 thread

per request

Each thread owns its read(). It blocks until bytes arrive. No async machinery anywhere.

~8 KB

kernel stack per thread

Plus a 1–8 MB virtual userspace stack.

1000s/sec

context switches under load

Every read() + every block + every wakeup is a kernel trip.

WHERE IT BREAKS

Scheduler contention

At 200+ concurrent threads, the kernel spends more time context-switching than doing your I/O. The CPU profile shows kernel time eating user time.

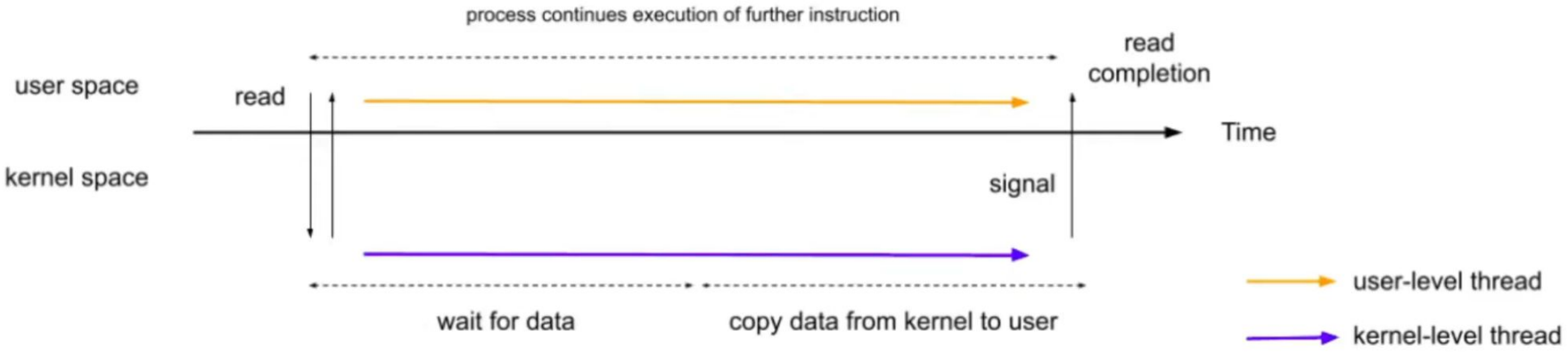
Lock contention

Any shared state — connection pools, output buffers, allocators — turns into a lock fight. More threads = worse contention, not more throughput.

FD pressure

Default ulimit is 1024. You can raise it. But every open fd costs kernel memory, and the kernel's fd lookup tables get slower at scale.

Asynchronous I/O



epoll: one thread, many sockets

The modern Linux event loop.

2002 Linux 2.5.44. “The interface that made C10K possible.” *predecessor to io_uring*

1 thread

monitoring N sockets

Many fds, one event-loop thread. epoll_wait() returns the set of fds that are ready.

2 syscalls

per operation, minimum

epoll_wait() to find a ready fd. Then read() (or recv()) to actually pull bytes. Every. Single. Operation.

C10K → C10M

what it enabled, what it can't

epoll cleared the way to 10,000 concurrent connections on commodity hardware. Getting to 10 million needs less kernel boundary.

epoll: one thread, many sockets

The modern Linux event loop.

2002 Linux 2.5.44. “The interface that made C10K possible.” *predecessor to io_uring*

1 thread

monitoring N sockets

Many fds, one event-loop thread. epoll_wait() returns the set of fds that are ready.

2 syscalls

per operation, minimum

epoll_wait() to find a ready fd. Then read() (or recv()) to actually pull bytes. Every. Single. Operation.

C10K → C10M

what it enabled, what it can't

epoll cleared the way to 10,000 concurrent connections on commodity hardware. Getting to 10 million needs less kernel boundary.

WHERE IT BREAKS

Syscall floor matters at scale

At low concurrency, 2 syscalls per op is fine. At hundreds of thousands of ops/sec, each syscall (~100 ns plus side effects) becomes a measurable fraction of total CPU.

Edge-triggered mode

Default level-triggered mode is easy but inefficient. Edge-triggered is faster but you must drain each fd fully or events vanish. Most real epoll bugs live here.

Still synchronous at the read

epoll tells you bytes are ready. The read() that follows is still a synchronous non-blocking syscall.

POSIX AIO: the failed attempt

Linux's previous attempt at async I/O.

1995 POSIX.1b. "The async I/O nobody actually uses."

submit → done

the right interface

aio_read() returns immediately. You get notified via signal or callback when bytes arrive. On paper, exactly what you want.

thread pool

what's actually underneath

On Linux, a userspace thread pool issues blocking reads behind the scenes. All the cost of threads, none of the API benefit.

O_DIRECT only

the kernel-native variant

There's also io_submit/io_getevents. Truly async for O_DIRECT block I/O only.

WHY IT FAILED

Half the API works

Submit a regular file read? Sometimes async, sometimes blocking.
Submit a socket read? Always blocking.
Predicting behavior needs the fd type and the kernel version.

No batching

Even when async, each aio_read() is its own syscall. No way to submit a batch of operations with one kernel crossing.

Nobody adopted it

The community wrote epoll reactors instead and waited for something better.

io_uring

- io_uring is a Linux kernel system call interface for storage device asynchronous I/O operations (interface like *read()/write()* or *aio_read()/aio_write()*)

Two ring buffers shared between userspace and the kernel: one for I/O requests, one for completions. The kernel does the work asynchronously; you don't make a syscall per operation.
Introduced by Jens Axboe in kernel 5.1 (2019).
Effectively the default for high-throughput I/O on modern Linux.

WHY IT EXISTS

POSIX AIO failed

- POSIX aio_read was the previous attempt
- Linux implementation fell back to threads
- Nobody used it; epoll won by default

HOW IT WORKS

Two shared ring buffers

- Submission Queue (SQ): you write requests
- Completion Queue (CQ): kernel writes results
- io_uring_enter() processes a whole batch at once

WHY IT'S FAST

Almost no syscalls

- Batch submission: 200 ops, 1 syscall
- Optional SQPOLL mode: kernel polls, zero syscalls
- Per-op cost approaches a memory store

The two rings

WHAT IT IS

Two shared-memory ring buffers. You write requests to one, the kernel writes results to the other. The kernel does the work asynchronously, so one thread keeps hundreds of operations outstanding.

FOUR THINGS IN PLAY

SQE

submission queue entry

SQ ring

shared with the kernel

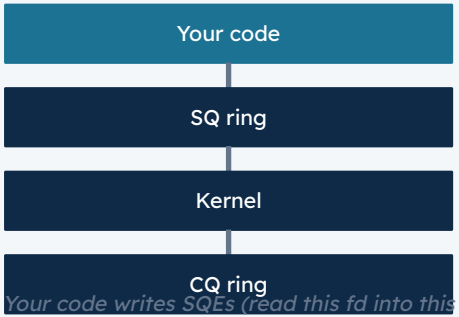
CQ ring

kernel writes here

CQE

completion queue entry

HOW DATA FLOWS



Your code writes SQEs (read this fd into this buffer). Kernel reads SQ asynchronously, does the work, writes CQEs. You harvest completions when ready.

WHY THIS MATTERS

1 syscall

to submit a whole batch of operations

0 syscalls

with SQPOLL — kernel thread polls the ring

Zero copy

buffers can be pre-registered with the kernel

SQE → CQE

BEFORE

read()

Before io_uring, this was all we had:
Each call = one syscall, blocking.

A read() returns when data arrives.
We scaled up via threads.

WITH io_uring

ring buffers

Our code prepares operations into the SQ ring, then *Submit many at once, harvest later*.
The kernel does the I/O asynchronously, batched, in the background.

1 **Prep**
io_uring_get_sqe() to claim a slot, io_uring_prep_read() to describe the op

2 **Submit**
io_uring_submit() — one syscall pushes the whole batch to the kernel

3 **Wait + Harvest**
io_uring_wait_cqe() blocks until a completion; loop until you've drained the CQ ring

Key Operations

OP 1

READ

Read N bytes from an fd into a buffer. The bread-and-butter op — equivalent to a non-blocking read() but submitted in batch.

OP 2

RECV

Receive bytes from a socket. Same semantics as recv() but submitted async. The op AnyBlob uses for HTTPS response bodies.

OP 3

POLL_ADD

Wait for a fd to become readable/writable, then complete. Effectively a one-shot epoll inside the ring. The op libcurl uses when it asks io_uring to be its polling backend.

OP 4

OPEN_AT

Open a file by path, async. Saves you from blocking the thread on a filesystem op — useful when paths might trigger network filesystem lookups.

io_uring vs the previous attempts

WHAT CAME BEFORE

Threads + blocking, then epoll

Two-syscall per op.

Threads: one fd per thread, one read() per op.

epoll: one thread, many fds, but every op is poll-for-readiness + actual-read.
The kernel boundary is crossed twice.

Scales OK to thousands of connections, but at hundreds of thousands of ops/sec, syscall overhead becomes a measurable fraction of CPU.

WITH io_uring

Ring submission, batch completion

Near-zero syscall floor.

Submit 200 reads with one io_uring_submit().

Harvest completions from the CQ ring without entering the kernel at all.

With SQPOLL, even the submit is gone, a kernel thread polls the ring.

Scales to tens of thousands of in-flight ops on a single thread. Cost-per-op approaches that of a memory store.

io_uring isn't always faster, the win is in scaling

AnyBlob: saturate the NIC from S3

TUM Durner, Leis, Neumann. “Exploiting Cloud Object Storage for High-Performance Analytics.” *PVLDB 2023*

≈80 Gbit/s

achieved on a 100 Gbit/s NIC

Single c5n.18xlarge instance, no caching, raw GETs against S3.

200–250

concurrent outstanding requests

Hundreds of in-flight GETs are needed to saturate the NIC.

8–16 MiB

cost-throughput-optimal request size

Smaller → latency dominates. Larger → you pay per-MiB without speedup.

HOW THEY DID IT

io_uring

Async I/O instead of one-thread-per-request. No oversubscription.

Request hedging

Re-fire stragglers after a deadline. ~5% of requests are unusably slow.

DNS rotation

Cycle through endpoint IPs. Some S3 nodes are just better than others.

AnyBlob: three ideas that compose

What you'd build if you wrote an HTTPS client from scratch on top of io_uring

IDEA 1

Saturate the NIC, always

A 12.5 Gbps NIC costs the same to operate whether you push 100 Mbps or 12 Gbps. Anything less is leaving money on the table.

IDEA 2

Parallelism, not bandwidth-per-connection

TCP CWND × RTT caps a single connection at hundreds of Mbps. To reach gigabits per second you need many connections.

IDEA 3

io_uring as the substrate

Building a complete HTTPS client on io_uring primitives allows to manage thousands of in-flight requests with low overhead.

Lab 4: Reading from Cloud Storage

Your job:
- implement a *single-connection* baseline
- then concurrent fetchers using *std::thread*
- then the same concurrent fetchers using *io_uring*
- run them on a *cloud VM* in the same region as the storage account
- sweep *parallelism* and find out what your hardware iscapable of

VM Provisioning + io_uring programming guidelines in the README.

Deadline: **02.06, 23:59:59**

Questions

Thank you for your attention!