Practical Course: Cloud-Based Data Processing

Michalis Georgoulakis

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

12.05.2026

Agenda

Lab 3 is underway

- Deadline 19.05 13:59:59

Today

- Lab 2 Discussion
- Join Operations
- Cloud Storage

Next Week: First assignment in the Cloud

- Benchmarking and working with cloud storage

NFS input might not have been NFS

- `/mnt/taxi/` = local NVMe (image-baked, every node has own copy)
- `/mnt/taxi-shared/` = NFS export from node0

Goal: measure NFS vs local NVMe
file:///mnt/taxi-shared/filelist.csv ← ~4 KB, goes over NFS ✓
└─ contains URLs like:
file:///mnt/taxi/partition_0001.csv ← hardcoded LOCAL path

Workers resolve those URLs **on their own NVMe**. NFS only served the filelist file. The 4 GB of partition reads never touched the network.

You can detect with tools like `nfsstat -s` on node0, `iftop -i <NFS NIC>`
Fix: Rewrite every URL in the filelist to point at the NFS mount instead of local NVMe.

Measurement methodology is important: when a result surprises you, verify the bytes are actually flowing where you think.

Takeaways

- Network and disk are independent cost dimensions.
- Algorithms don't run on math, they run on hardware.
- Same code, swap input path can have 100× difference.
    - Where bytes physically live (NVMe / NFS / TUM) can be the single dominant performance factor.
- Pre-aggregation is a necessary design (local combiner).
- You can't reason about a distributed system without measuring it.

Experimental findings

- Shuffle pattern: map → hash-partition → exchange → merge ✅
- Local combiner shrinks shuffle to distinct-key cardinality 🟡
- Per-shard top-K is correct because shuffle key = agg key ✅
- Local NVMe > NFS at the input layer 🟡
- Peer-mode shuffle slightly ahead of NFS-mode shuffle for our use case 🟡
- Shard count is a U-curve; sweet spot ≈ 32-64 workers ✅
- Verify your measurement before trusting it 🟡

Refresher: Join algorithms (inner equi-joins)

Two high level strategies to quickly find tuples with the same attribute values: **sort**, **hash**

Nested loop join

For every row of the outer relation, scan the inner relation for matches.

Simple · works for any predicate · O(n·m) without indexes

Sort-merge join

Sort both inputs by the join key, then walk them in lockstep to emit matches.

Equality joins · sorted outputs · benefits from existing sort orders

Hash join

Build an in-memory hash table on one input; probe it with rows from the other.

Equality joins · usually fastest in memory · hash on the smaller side

Nested Loop Join

Naive, expensive but simple approach.

```
foreach tuple r ∈ R:
    foreach tuple s ∈ S:
        if match(r, s) then emit (r, s)
```

R(id, name)	S(id, value, cdate)
(600, "MethodMan")	(100, 2222, "10/6/2025")
(200, "GZA")	(500, 7777, "10/6/2025")
(100, "Andy")	(400, 6666, "10/6/2025")
(300, "ODB")	(100, 9999, "10/6/2025")
(500, "RZA")	(200, 8888, "10/6/2025")
(700, "Ghostface")	
(400, "Raekwon")	

Sort-merge join, in one slide

WHAT IT IS

A two-phase algorithm. Sort each input on the join key, then scan both sides in parallel with two cursors and emit matches as you go. Sequential I/O, and the output comes out sorted for free.

THE THREE PHASES

Sort, sort, merge

1. Sort R on the join key
2. Sort S on the join key
3. Merge: two cursors, one pass, emit matches

THE COST

O(n log n) per side

- Then a single O(n + m) linear merge
- Sequential reads throughout
- Output is sorted, saves an ORDER BY downstream

WHEN IT SHINES

When the sort is free

- Inputs come sorted from B-tree index scans
- The next operator needs sorted output anyway
- Data is too big for aggregate cluster RAM

Hash Join

Two phases: build the index, probe with the other side

PHASE 1

Build

Scan the smaller relation R. Hash each row by its join key and insert it into an in-memory hash table H.

```
for r in R:
    H[hash(r.key)].append(r)
```

PHASE 2

Probe

Scan the other relation S. For each row, look up its key in H and emit every matching pair.

```
for s in S:
    for r in H[hash(s.key)]:
        if r.key == s.key: emit(r, s)
```

PREDICATE

Equality only — needs key = key

COST

O(|R| + |S|) when H fits in memory

RULE OF THUMB

Hash on the smaller side

Distributed Hash Join Options

ONE SIDE SMALL

Broadcast join

Replicate the small side to every worker. The big side stays put.

Network: |small| × workers · Memory: small fits per worker

BOTH SIDES LARGE

Shuffle hash join

Hash-partition both sides by the join key, then run hash join locally on every node.

Network: nearly all of both inputs · Memory: per-partition hash table

ALREADY ALIGNED

Co-located join

If both sides are already partitioned by the join key, no shuffle is needed at all.

Network: zero · The cheapest possible distributed join

Disk Spilling

What if the data doesn't fit in memory?

When an in-memory data structure (a hash table, a sort buffer, an aggregation state) outgrows RAM, the engine writes the overflow to local disk and reads it back when needed. The query still runs. It just runs slower.

WHEN IT HAPPENS

Three usual suspects

- Hash join build side bigger than worker memory
- External sort over data larger than RAM
- Aggregation with too many distinct group keys

USED TO COST

Overkill on HDDs

- HDDs: ≈100 MB/s sequential, 10 ms seeks
- 100× slower than RAM
- One spilling join could ruin the whole query

COSTS NOW

Still bad, but bearable

- NVMe: 3–7 GB/s sequential
- Only 10–20× slower than RAM
- Modern engines plan for it as a failover approach.

Graefe 1994: dualities, not duels

The first systematic comparison — argued sort and hash were closer than anyone thought, and a DBMS needs both

1994 Graefe, Linville, Shapiro. "Sort vs. Hash Revisited." *IEEE TKDE 1994*

% not ×

the gap, when both are tuned

Costs differ by percentages, not factors — if both are implemented with equal care.

duals

merging ↔ partitioning

Every sort optimization has a hash counterpart — and vice versa.

ship both

in a real DBMS

Each algorithm has corner cases the other loses badly on — the optimizer picks.

THE TWO CORNER CASES

Different-sized inputs

Hash wins. Sort cost is dominated by the larger side; hash recursion depth depends only on the smaller one.

Skewed hash keys

Sort wins. Hashing partitions logically by value is punished by bad distributions. Sort divides physically, and is uniform either way.

Kim 2009: hash leads, but not for long

Multi-core Core i7 — hash still wins today, but project SIMD forward and the verdict flips

2009 Kim, Sedlar, Chhugani et al. “Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs.” *PVLDB 2009*

100M tup/s

hash join throughput

17× faster than the best prior CPU result, 8× faster than reported GPU numbers.

47–80M

sort-merge throughput

Hash still leads on 128-bit SSE — by roughly 2× at large inputs, even with SIMD-optimized merge networks.

≥512-bit

SIMD width to flip the verdict

Project SIMD forward — bitonic sort-merge overtakes hash. The tide hasn't turned, but it's about to.

TWO TRENDS FAVOR SORT-MERGE

Wider SIMD

Bitonic merge networks scale linearly with SIMD width. Hash join scatter/gather doesn't.

Bandwidth per core shrinking

Hash needs ~1.5× more memory traffic than sort-merge: partition, then probe. As per-core BW falls, that ratio bites first.

The forecast

When 512-bit SIMD arrives, sort-merge will be the faster algorithm. AVX-512 was already announced. Wait a few years.

Balkesen 2013: hash holds the line

2013 Balkesen, Alonso, Teubner, Özsu. “Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited.” *PVLDB 2013*

2× faster

radix-hash vs. sort-merge

On 256-bit AVX, 64 threads, NUMA-aware — at hundred-million-tuple inputs.

billions

where the gap finally closes

Sort-merge only catches up when inputs reach billion-tuple scale and partition passes start to matter.

both gained

hardware lifted everything

Wider SIMD helped sort-merge — but the same advances also helped radix-hash. The ranking didn't change.

WHY THE PREDICTION MISSED

Merge trees don't fit

Kim assumed merge trees fit in last-level cache. With 64 threads sharing L3, they don't — multi-way merge degrades fast at high fan-in.

Hash got faster too

Software-managed buffers let radix-hash do in one partition pass what previously took two. Sort-merge wasn't racing a stationary target.

Sort-merge's surviving niche

Billion-tuple inputs, or downstream needs sorted output. Otherwise — same verdict as Graefe and Kim — radix-hash wins.

Where sort-merge still wins

CASE 1

Already-sorted inputs

If both sides arrive sorted on the join key — e.g., B-tree index scans in OLTP — the sort cost is zero and merge is essentially free.

seen in: Postgres, MySQL, SQL Server · index nested loops, transactional joins

CASE 2

Data > aggregate cluster RAM

External sort scales gracefully past memory. Hash join either spills (more I/O) or fails. Rare in practice today — RAM is plentiful.

seen in: Massive historical scans · data-warehouse appliances of the 2000s

CASE 3

Downstream needs sorted output

If the next operator is an ORDER BY, window function, or merge-style aggregation, sort-merge produces sorted output for free.

seen in: Some streaming systems · certain analytic window queries

Today: Four trends

REASON 1

Memory got cheap

Cloud workers have tens or hundreds of GB of RAM. The hash table usually fits in memory.

in your labs: This makes BHJ a viable operation.

REASON 2

Disks got fast

NVMe makes disk spilling cheap. Hash join's worst case (spill to disk) is not as disastrous as it used to be.

in your labs: Lab 3 shuffle write: partitions stream to local disk between phases.

REASON 3

Hashes got better

Modern hash functions (xxHash, CRC32) are vectorized, low-collision, and fast enough that hashing is no longer the bottleneck.

in your labs: Lab 2 + 3 partitioner: hash-by-key is a tight inner loop.

REASON 4

Parallelism got easy

Partitioned hash tables parallelize trivially: one table per shard, no synchronization across shards.

in your labs: Lab 3 architecture: shards are independent. Sort-merge can't say the same.

To hash or to sort: the end of the debate?

- Some modern analytical engines (DuckDB, CedarDB, Velox, your Lab 3) don't even ship SMJ.
- Corner cases (sorted inputs, ORDER BY downstream) are real but niche enough for a hash + re-sort to win.
- OOM Performance
- Still, both can be found in most industry systems (Spark still prefers SMJ by default.

Spark SQL: 10 megabytes

Why does the most-used analytics engine set its broadcast threshold so low?

THE DEFAULT

spark.sql.autoBroadcastJoinThreshold = 10 MB

Below 10 MB, Spark broadcasts. Above 10 MB, it shuffles.

WHAT YOU MEASURED

Your Lab 3 number

Working alone on a quiet cluster, broadcast may win much further out.

Single query · full cluster memory · no contention

WHAT SPARK SHIPS

Spark's number

10 MB. That's the bound where broadcast still wins in a real, shared, multi-tenant production environment.

Many queries · shared memory budget · defensive default

Why 10 MB and not 200 MB?

PICTURE THIS

A 50-worker Spark cluster running 8 analytical queries concurrently

200 MB	× 50	× 8	= 80 GB
broadcast table	*executors*	*concurrent queries*	*memory pressure*

SPARK ASSUMES

Potentially noisy cluster

Many queries, shared executors, unknown concurrent load. A 200 MB broadcast could OOM the whole tenant. 10 MB is the safe floor.

The cost-optimal threshold isn't a property of the algorithm, but it's a property of the deployment.

Statistics-driven vs. analytical choice

Catalyst's approach vs. the closed-form formulas you derived for Lab 3

PRODUCTION · CATALYST

Statistics-driven

Cost is computed from collected stats: row counts, distinct-value estimates, recently added histograms.

needs	ANALYZE TABLE COMPUTE STATISTICS
good at	queries on tables you've actually seen
fails on	fresh tables, ad-hoc files, S3 dumps

YOUR LAB 3

Analytical

Cost is a closed-form formula in cluster parameters: N, M, P, B. No measurements required.

needs	a model of the cluster
good at	predicting before you run anything
fails on	skew, stragglers, real noise

Baldacci & Golfarelli (Bologna) built a hybrid — Spark physical plan + cluster params + DB stats. ~20% error on a 1000-query benchmark.

Where does data live?

CLOSED SYSTEM

Postgres, MySQL, classical OLAP

The DBMS owns the bytes.

Data lives in its own pages, on its own disk, in its own format. The DB controls layout, sort order, compression, statistics.

Sort-merge can rely on inputs being sorted. Hash can pre-partition. The optimizer trusts what it sees.

OPEN SYSTEM

DuckDB on S3, Spark, your Lab 4

The bytes live elsewhere.

CSV, Parquet, JSON on S3 / Azure / GCS. No structural invariants. No guaranteed sort. Layout is whatever the upload pipeline produced.

Sort-merge has nothing to rely on. Hash works if you can get the bytes off the wire fast enough.

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

Object storage looks slow

Each request takes 30–150 ms before the first byte. That's a lot by NVMe standards.

FIRST BYTE LATENCY

≈30 ms

Time from request to first byte.

PER-OBJECT BANDWIDTH

50–95 MiB/s

Per-request bandwidth, even for big objects. S3 caps the per-stream rate.

NVMe COMPARISON

≈3–7 GB/s

What local NVMe comfortably gives you. Roughly 100× the per-stream cloud rate.

THE NAIVE CONCLUSION

"Cloud object storage is too slow for analytics. We have to cache everything to local SSDs."

THE ACTUAL CONCLUSION

Per-request bandwidth is low. *Aggregate bandwidth, with enough concurrent requests, might work.*

The data can be accessed fast. The API for asking for one byte at a time can not.

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.

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.

Every cost model assumes bytes are free

THE IMPLICIT ASSUMPTION

“Once I've decided to read a table, the bytes show up at memory bandwidth. I only need to count joins, shuffles, and writes.”

THE REALITY ON CLOUD STORAGE

One request → 50 MiB/s.
Hundreds of concurrent requests → 10 GiB/s.
Concurrency should now be part of a cost model

NEW TERMS YOUR COST MODEL WILL NEED

Per-request latency
30–150 ms before the first byte

Request concurrency
How many in-flight GETs?

Prefetch depth
How far ahead do you read?

Lab 4: from /mnt/taxi to real cloud

Until now you've been reading from a local mount. Next week, the data really lives in Azure Blob.

WHAT YOU'LL BE WORKING ON

1 **Range reads**
Pull byte ranges of large objects — not whole files

2 **Concurrency**
Many outstanding GETs, async I/O

3 **Smarter prefetch**
Read ahead so processing never waits

Questions

Thank you for your attention!