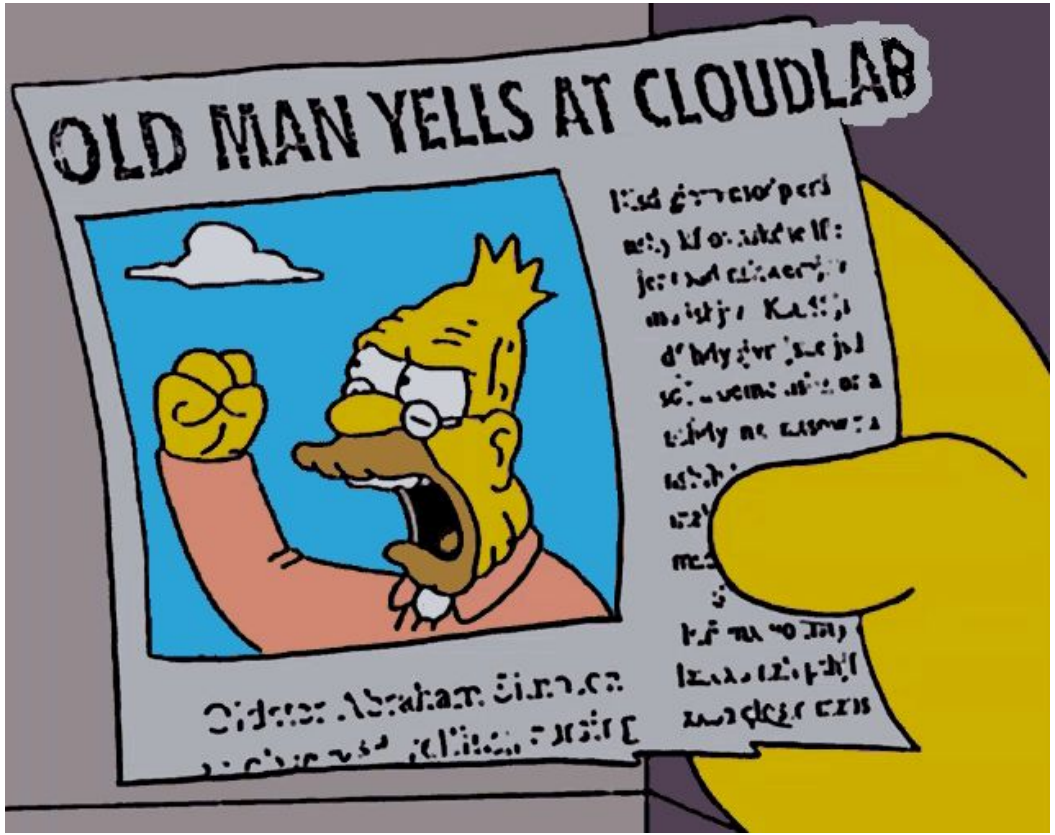
Practical Course: Cloud-Based Data Processing

Michalis Georgoulakis

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

05.05.2026

Lab 2 is ~~over~~ extended until Thursday 23:59:59



Agenda

Lab 2 Submission Window

- Extended to 07.05 23:59:59

Today

- Lab 2 Retrospective
- Join operations
- Lab 3 Introduction
- Azure Setup

Next Week: Second Assignment Q&A

- We will discuss it in more detail
- Be prepared to present your solution

Shared Storage: root cause + fix

Initial observation: accessing shared dataset was notoriously slow

Root cause

- /mnt/taxi was an iSCSI mount from a shared TrueNAS storage server on CloudLab.
  - block storage exposed over a TCP network
- Packet loss fairly high → TCP waits before resending, doubling its wait each retry (≈ 1 s → 2 s → 4 s). One lost packet can stall a single read for 5–7 seconds.
- The more workers read concurrently, the worse it got, both the fabric and the storage server are *shared with other CloudLab tenants*.

Fix - already in profile

- Dataset is now baked into the OS image.
- Each node boots with its own copy of /mnt/taxi on local NVMe.

Skew - When one shard does all work

Hash partitioning spreads keys evenly across shards. But the rows aren't evenly distributed across keys.

- NYC taxi: 265 PULocationIDs, but Manhattan zones get ~80% of pickups.
- JFK alone (id 132) makes its shard ~10× heavier than the average.
- One slow shard sets the wall-clock time for the whole job.

What one can do about it:

- Salting: re-hash hot keys with (key, random % N); re-aggregate after.
- Adaptive execution (AQE): detect hot partitions at runtime, split them on the fly.

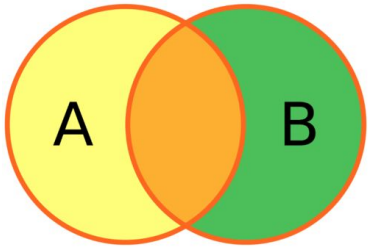
From Lab 2 to Lab 3

#	Pattern	Communication	Lab
1	Embarrassingly Parallel	None	HTTP fetch in Lab 1
2	Reduce / fold	All workers → coordinator (one shot)	Lab 1 (complete_partition merge)
3	Shuffle-aggregate	Workers → workers, hash-partitioned	Lab 2
4	Shuffle-join (New)	Both inputs → workers, co-partitioned	Lab 3

What changes when there are two inputs?

What is a join?

A join combines records from two or more datasets by matching values in a shared attribute, the **join key**.



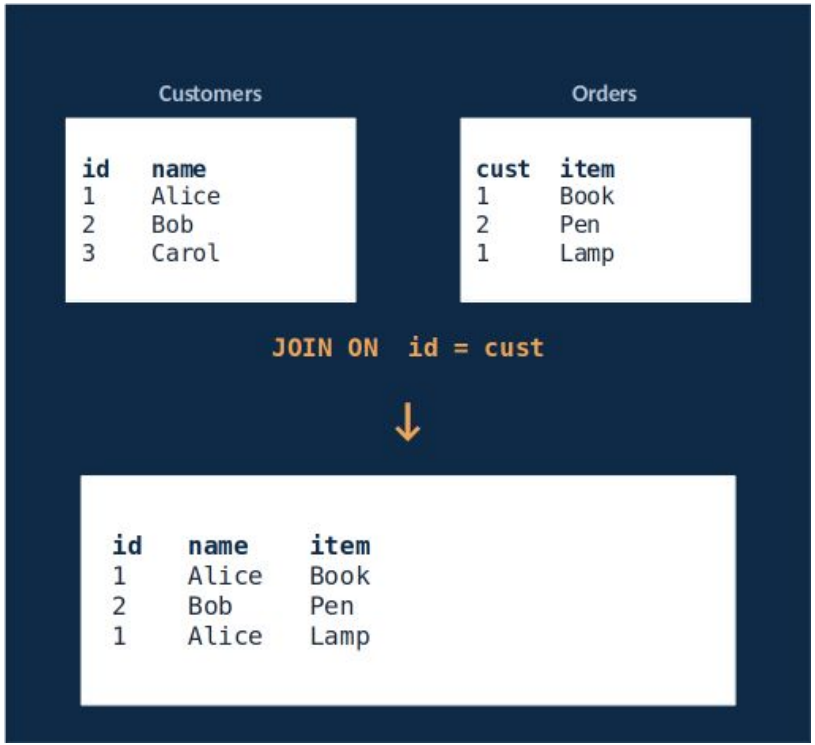
Where you see it:

- *Relational databases*: SQL JOIN over tables
- *DataFrames*: pandas merge, R dplyr left_join
- *Big-data engines*: Spark, Flink, MapReduce shuffles
- *ETL & data integration*: linking customers, orders, events

A join clause in the Structured Query Language (SQL)

```
SELECT COUNT(*)
  FROM orders, lineitem
 WHERE o_orderkey = l_orderkey
```

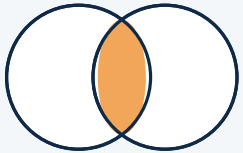
What is a join?



Kinds of join

MOST COMMON

Inner join



Keeps only rows that have a match in both tables.

Unmatched rows on either side are dropped.

PRESERVE UNMATCHED

Outer join

LEFT · RIGHT · FULL JOIN

Keeps unmatched rows from one or both sides.

Missing values from the other side are filled with NULL.

LEFT keeps all of A · RIGHT keeps all of B · FULL keeps both

SAME TABLE TWICE

Self join

employees

name	manager
Alice	–
Bob	Alice
Carol	Alice

row → row

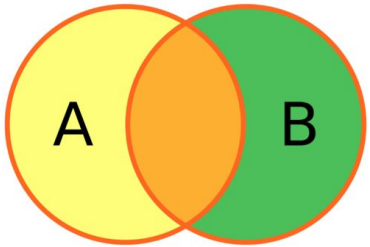
A table joined with itself, using two aliases.

Used for hierarchies (employee ⋈ manager) or comparing rows within the same table.

Joins in relational databases

SQL is declarative: the user says what, the system decides how

```
SELECT c.name, o.item, p.price
FROM   customers c
JOIN   orders   o ON c.id = o.cust
JOIN   products p ON o.pid = p.id
WHERE  p.price > 50;
```



- **Commutative**: A ⋈ B = B ⋈ A. Order of operands does not change the result.
- **Associative**: (A ⋈ B) ⋈ C = A ⋈ (B ⋈ C). Grouping does not change the result.
- **Cost-sensitive** — Plans for the same query can vary by orders of magnitude.

The query optimizer’s two freedoms

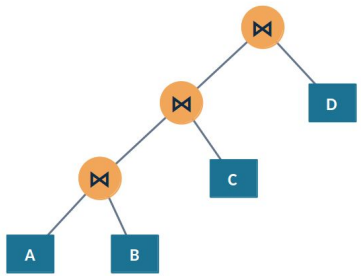
Two independent choices that define different execution plans.

Join Order
Which tables to combine first.

Join Method
Which algorithm to run for a join.

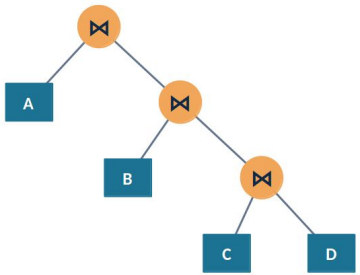
Left-deep

Inner operand is always a base table



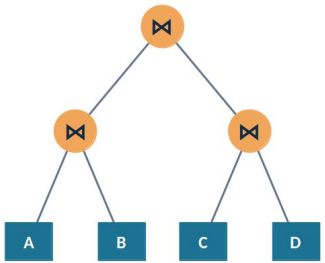
Right-deep

Outer operand is always a base table



Bushy

Both operands can themselves be joins



Join algorithms

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

Hash Join, Mechanically

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: Joins and Repartitioning

Joins are often the most expensive distributed operation

- Matching rows from both tables must be co-located on the same worker.

Requirement: Repartition both input tables on the join key

Example: Joining `customer.id = order.customer_id`

- All rows for a given customer must end up in the same partition.

Repartitioning = Shuffle

- Both tables hashed by join key
- Data is redistributed across the cluster
- Partitions are written to disk before the join

Local / Co-partitioned Join (Best Case)

If both tables are already partitioned the same way:

- Same partitioning key
- Same number of partitions
- Same hash function / distribution boundaries

Then:

- No shuffle
- No broadcast
- Each corresponding partition joined locally
- Ideal performance

This is why bucketing / clustering / distribution keys exist.

Repartition (common case)

Matching rows can only meet if they live on the same node

BEFORE — arbitrary partitioning

NODE 1	NODE 2	NODE 3
R: { a, e, h }	R: { d, b, i }	R: { c, f, g }
S: { b, f, i }	S: { a, e, g }	S: { d, c, h }

SHUFFLE · repartition both R and S by hash(join_key)

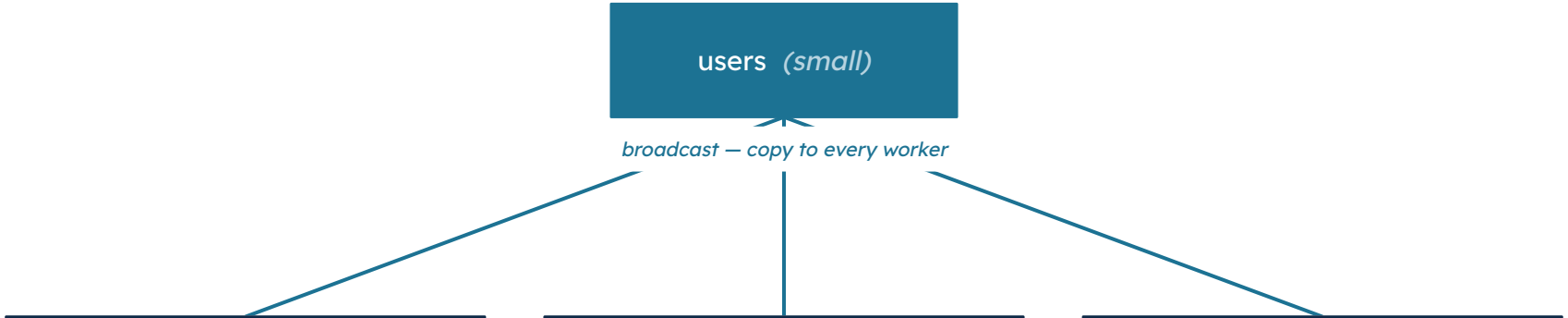
AFTER — co-located by join key

NODE 1 · KEYS a–c	NODE 2 · KEYS d–f	NODE 3 · KEYS g–i
R: { a, b, c }	R: { d, e, f }	R: { g, h, i }
S: { a, b, c }	S: { d, e, f }	S: { g, h, i }

The cost: nearly all of R and all of S move across the network — usually the dominant cost of the join.

Broadcast Join - Eliminating the shuffle

When one side is small, replicate it everywhere instead of moving the big side



WHAT MOVES

Only the small side · the big side never leaves its node

WHEN IT PAYS

|small| × workers ≪ full shuffle of both sides

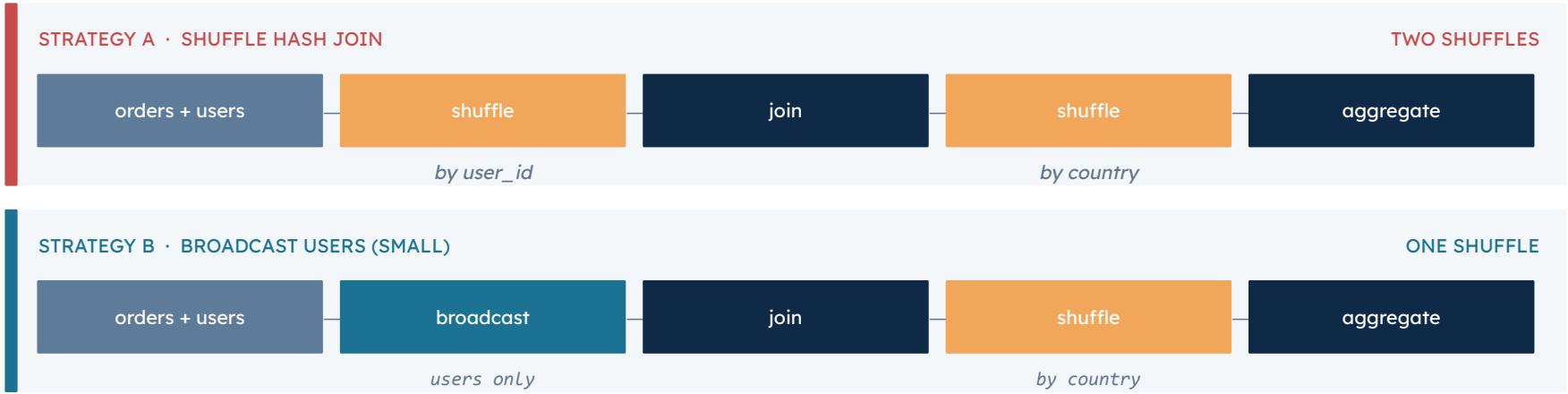
WHEN IT BREAKS

Small side no longer fits in worker memory

When the keys don't line up

```
SELECT u.country, COUNT(*) FROM orders o JOIN users u ON o.user_id = u.id
GROUP BY u.country;
```

join key: user_id · *aggregation key:* country



Why broadcast could win: the join's shuffle would have partitioned by user_id, useless for the next GROUP BY country. Skipping it leaves only the aggregation shuffle.

Choosing a strategy

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

WATCH THE NEXT KEY

Mind the pipeline

If the downstream operation needs a different key, you'll shuffle anyway. B

Optimize the whole plan, not each operation in isolation

Operator Selection: Distributed Joins

Distributed engines consider:

- Table sizes **(cardinality estimation)**
- Existing partitioning / bucketing
- Network I/O cost **(cost model)**
- Memory availability (for broadcast)
- Join type and selectivity

And choose the cheapest **(plan enumeration, cost model)**:

- Local join (best but rare)
- Broadcast join (good when one relation is small)
- Shuffle (repartition) join (most expensive but scalable)

Join order + join strategy are now network-aware, not just CPU-aware.

Joins are a hot research topic

Foundation Work

IBM — Selinger, Astrahan, Chamberlin, Lorie, Price. **"Access Path Selection in a Relational Database Management System."** *SIGMOD 1979*

MICROSOFT — Graefe. **"Query Evaluation Techniques for Large Databases."** *ACM Computing Surveys 1993*

IBM — Swami, Schiefer. **"On the Estimation of Join Result Sizes."** *EDBT 1994*

Join order & cardinality estimation

TUM — Leis, Gubichev, Mirchev, Boncz, Kemper, Neumann. **"How Good Are Query Optimizers, Really?."** *PVLDB 2015*

TUM — Fender, Moerkotte, Neumann, Leis. **"Effective and Robust Pruning for Top-Down Join Enumeration Algorithms."** *ICDE 2012*

TSINGHUA + WISCONSIN — Zhao, Su, Yang, Yu, Koutris, Zhang. **"Debunking the Myth of Join Ordering: Toward Robust SQL Analytics."** *SIGMOD 2025*

Parallel & NUMA-aware execution

TUM — Bandle, Giceva, Neumann. **"To Partition, or Not to Partition, That is the Join Question in a Real System."** *SIGMOD 2021*

WISCONSIN — Blanas, Li, Patel. **"Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs."** *SIGMOD 2011*

ETH ZURICH — Balkesen, Teubner, Alonso, Özsu. **"Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware."** *ICDE 2013*

Lab 3: The Distributed Join Problem

Given a join query and a cluster shape, do you broadcast the small side or shuffle both sides?

Shuffle hash join

Hash-partition both sides on the join key. Build + probe locally on every shard.

Broadcast hash join

Replicate the small side to every worker. The big side never leaves its node.

TASKS

1. **Implement** — *both algorithms on the Lab 2 coordinator/worker skeleton*
2. **Measure** — *their runtime across 4 queries on real NYC Taxi data*
3. **Derive** — *a cost model from first principles*
4. **Predict** — *with a heuristic that picks the right algorithm*

Deadline: 19/05

Azure - our cloud for Block B

Azure will be the cloud vendor that we use for our “real” cloud experiments.

- $100 in Student credits
- No need to enter credit card details
- Same account everywhere (TUM / Microsoft 365 / Azure)

What we will use it for:

- **Labs 4-6:** Exploring cloud storage, networking, and compute

Setup guide: GitLab

Questions

Thank you for your attention!