Practical Course: Cloud-Based Data Processing

Michalis Georgoulakis

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

16.06.2026

Schedule

	Content	Lab Start	Lab End	Oral Discussions
16.06	Benchmarking in the Cloud	Project	Lab 6	Lab 4, Lab 5
23.06	Topic Selection			Lab 6
30.06	Q&A			
07.07	Q&A			
14.07	Project Presentations			

Lab 4: Cloud Storage

Goal: how fast can we read from Azure Blob Storage? **Theoretical Standard_B2 ceiling ≈ 700 MB/s.**

Two models

- **std::thread**
- **io_uring + curl_multi**

Four access patterns

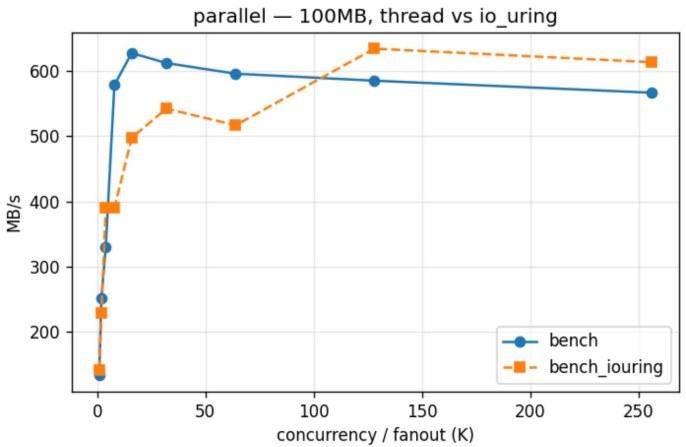
Sequential: one connection, one object
parallel: N concurrent full-object GETs
range: one object split into K byte-range GETs
tinyfanout: K tiny 4 KB range GETs

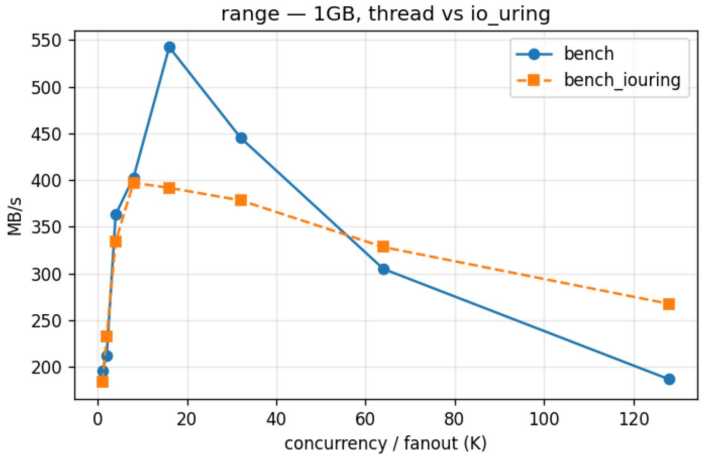
Lab 4: Findings per Access Pattern

Bandwidth-bound vs latency-bound — and the 4-connection cap shapes every curve.

Pattern	What binds it	Who wins	Key finding
sequential	Window / RTT (BDP)		One flow can't fill the NIC; larger objects increase throughput.
parallel	Link / aggregate window	threads (peak BW)	io_uring held back by MAX_HOST_CONNECTIONS = 4.
range	Same, but one object	parallel ≥ range	Splitting one object plateaus at the 4-conn cap.
tinyfanout	Handshake + RTT (latency)	io_uring, decisively	Warm-pool reuse wins; threads hit a ceiling and starve.

Lab 4: Findings per Access Pattern





	cross-region	in-region	factor
sequential 1 GB	82	172	2.1×
io_uring parallel @K=128	~100 (declining)	634	6×
io_uring tinyfanout peak	129 rps	317 rps	2.5×

Lab 4: Protocol & io_uring

HTTP/1.1 vs HTTP/2, and Range

- Keep-alive reuses a connection serially (one request in flight). HTTP/2 multiplexes many streams.
- Range Requests → 206 Partial Content.
- Azure Blob's REST endpoint is HTTP/1.1-compatible only only

How we actually used io_uring

Only as a readiness poll backend for curl_multi (a drop-in epoll).

- libcurl still owns recv/send and TLS. Operation = readiness; delivery = completion.

What did it buy us? Was it necessary?

Not strictly. The win over threads is the event loop (epoll would match it).

- io_uring's edge is syscall batching; the 4-conn cap and one-CQE-per-wait loop don't really amortize it.

Lab 4: io_uring + curl_multi Code Flow

1 Setup

- io_uring_queue_init(4096); curl_multi_init().
- Register socket_cb / timer_cb; set MAX_HOST_CONNECTIONS = 4.
- Build EasyCtx + curl_multi_add_handle.

2 Prime

- curl_multi_socket_action(CURL_SOCKET_TIMEOUT, 0).
- Fires socket_cb (“watch these fds”) and timer_cb (“wake in N ms”).
- still_running becomes the loop condition.

3 Event loop

- Arm one POLL_ADD per socket → io_uring_submit → wait_cqe_timeout.
- -ETIME: tell curl the timeout fired.
- Completion: drain all CQEs, decode fd, map to CURL_CSELECT_*.
- curl_multi_socket_action — libcurl does the real recv/send + TLS.

4 Teardown

- When still_running == 0, drain CURLMSG_DONE for errors.
- Sum body.size() across contexts; report throughput.
- Remove handles, cleanup multi + ring.

io_uring: Security Considerations

Why it worries defenders

- **Evasion / EDR blind spot:** async SQEs perform read/write/sendmsg/recvmsg in-kernel, so syscall hooks may never fire.
- **Large new attack surface:** novel kernel paths, replacing battle-tested code.
- **Hard to sandbox:** no fine-grained seccomp; functionality is opaque to BPF sandboxes, so it's effectively all-or-nothing.
- **Disabled for unprivileged users:** across Google's own production fleet / Google Cloud, they disabled it for unprivileged users stated they limit its use because they couldn't make it safe enough where untrusted code runs.

60%

of kernel exploits in Google's 2022 bounty targeted io_uring — since disabled by default in some environments.

~$1M

paid out by Google for io_uring vulnerabilities. Sandboxing via BPF isn't impossible and is on the roadmap.

Lab 5: Shuffle Substrates

A distributed shuffle is a **network workload** and the substrate you route it through is a **design choice**, not "the network."

Substrate	Equivalent	The trade-off
P2P TCP	Spark BlockManager, Lab 2/3	Fastest, but coupled — both ends live; no durability; N^2
Managed Redis	Memcached / dedicated Redis	Fast, decoupled — but volatile, RAM-bound, billed while idle
Storage Queue	Kafka / Kinesis / RabbitMQ	Decoupled + durable many→one; per-message overhead, size cap
Blob storage	S3 / GCS shuffle, Lab 4	Durable, elastic, cheap-at-rest — but high per-request latency

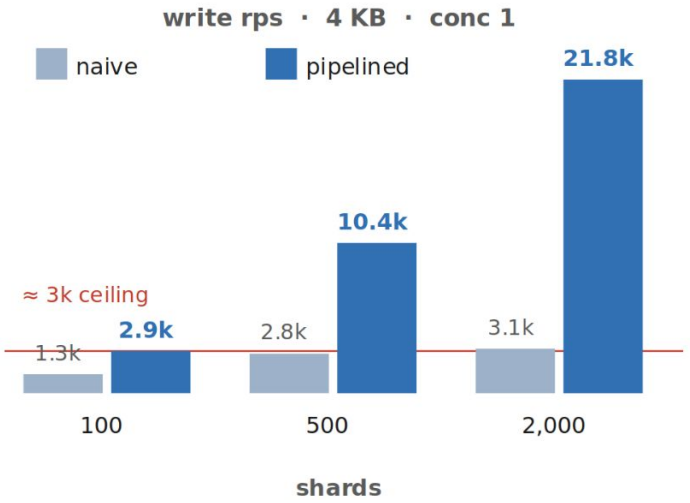
Lab 5: Findings per Substrate

No substrate wins everywhere

Substrate	Wins at	Watch out for
P2P TCP	Lowest latency; bulk streaming (1 RTT)	Coupled, fragile, no durability; N^2 connections
Redis	Fast small/medium — if pipelined	Naive = N×RTT; volatile; RAM-bound; billed idle
Storage Queue	Decoupled many→one work distribution	64 KB cap + base64; ≥3 ops/msg; at-least-once → dedupe
Blob	Durable, elastic, cheap bulk transfer	High Time-To-First-Byte; small-shard read amplification

Lab 5: Pipelining Unlocks Redis

Shards	naive rps	pipelined rps	speedup
100	1,330	2,940	2.2×
500	2,750	10,450	3.8×
2,000	3,100	21,800	7.0×



Naive Redis is RTT-bound at ≈3,000 rps (≈1/RTT at the ~0.3 ms intra-region round-trip). Pipelining (window W=512) pays the round-trip once per batch, not once per shard.

Lambada: The Serverless Shuffle → Lab 6

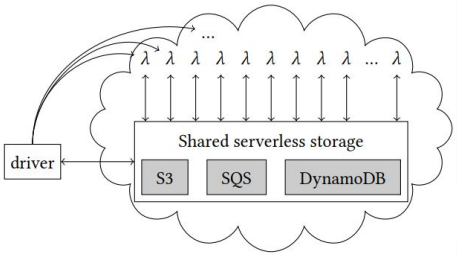
Three decoupled serverless stores (chosen by role & size)

S3: bulk data and the shuffle itself.

DynamoDB: small hot data.

SQS: short messages / coordination.

All scale to zero when idle — the bulk exchange runs on S3, not the key-value store.



The exchange operator is request-count-bound

A naive S3 shuffle needs $O(P^2)$ requests; at 256 workers the request cost already exceeds the worker cost.

Fix: multi-level exchange + write-combining → sub-linear requests. “Request count is the currency.”

The bridge to Lab 6: a serverless shuffle is only possible because storage decouples producers from consumers in time and identity.

Cloud Analytics Benchmarking

Benchmarking in the Cloud: Why It Is Different

- On-prem benchmarks (TPC-H, TPC-DS) measure **one engine on fixed hardware**: “which engine is fastest?”
- The cloud breaks that frame: resources scale **elastically**, services are **multi-tenant**, you may pay **per use**.
- The user-centric question becomes **cost (€) and latency**
  - not runtime on a fixed box.
- A cloud benchmark must capture elasticity, multi-tenancy, and handle different types of loads.

So: what does a **realistic** cloud workload even look like: and how do we benchmark it?

Existing Workload Telemetry Traces

Snowset (Snowflake)

Arrival Time	Exec. Time	CPU Time	Data Scanned	…
19:02	6 sec	50 sec	2 GB	…
19:04	2 sec	25 sec	100 MB	…
…	…	…	…	…

Redset (AWS Redshift)

Arrival Time	Was_ cached	# of scans	# of joins	…
19:02	Yes	7	6	…
19:04	No	3	2	…
…	…	…	…	…

Existing Workload Telemetry Traces

Snowset (Snowflake)

Arrival Time	Exec. Time	CPU Time	Data Scanned	…
19:02	6 sec	50 sec	2 GB	…
19:04	2 sec	25 sec	100 MB	…
…	…	…	…	…

Redset (AWS Redshift)

Arrival Time	Was_ cached	# of scans	# of joins	…
19:02	Yes	7	6	…
19:04	No	3	2	…
…	…	…	…	…

Can we get meaningful workloads out of these traces?

The Cloud Analytics Benchmark (CAB)

- Real workloads differ from TPC-H/DS: more variety in CPU/IO load, more **DML** (in-warehouse ETL), bursty multi-tenant arrivals.
- van Renen & Leis, PVLDB 2023: first to build a cloud benchmark based on an industry **trace** (Snowflake's Snowset, ≈69 M queries).
- CAB = TPC-H schema + per-tenant query streams + realistic arrival patterns; scored on **query latency and € per run**.

But the trace holds only **statistics: not the SQL**.

The Cloud Analytics Benchmark (CAB)

- Real workloads differ from TPC-H/DS: more variety in CPU/IO load, more **DML** (in-warehouse ETL), bursty multi-tenant arrivals.
- van Renen & Leis, PVLDB 2023: first to build a cloud benchmark based on an industry **trace** (Snowflake's Snowset, ≈69 M queries).
- CAB = TPC-H schema + per-tenant query streams + realistic arrival patterns; scored on **query latency and € per run**.

But the trace holds only **statistics: not the SQL**.

- We are still running the same TPC-H queries as before, but also testing *concurrency* and *elasticity*.

Matching-Based Query Synthesis

Telemetry Dataset (unknown queries)

CPU Seconds	Data Scanned	Arrival Time
30	5	3
15	7	7

Known SQL Pool

Query	CPU Seconds	Data Scanned
Q1	30	5
Q2	5	3
Q3	10	4

Synthesized Dataset

Telemetry Query #	Target CPU Time	Target Data Scanned	Matched Query/-ies
1	30	5	Q1
2	15	7	-

Matching-Based Query Synthesis

Telemetry Dataset (unknown queries)

CPU Seconds	Data Scanned	Arrival Time
30	5	3
15	7	7

Known SQL Pool

Query	CPU Seconds	Data Scanned
Q1	30	5
Q2	5	3
Q3	10	4

Synthesized Dataset

Telemetry Query #	Target CPU Time	Target Data Scanned	Matched Query/-ies
1	30	5	Q1
2	15	7	-

Matching-Based Query Synthesis

Telemetry Dataset (unknown queries)

CPU Seconds	Data Scanned	Arrival Time
30	5	3
15	7	7

Known SQL Pool

Query	CPU Seconds	Data Scanned
Q1	30	5
Q2	5	3
Q3	10	4

Synthesized Dataset

Telemetry Query #	Target CPU Time	Target Data Scanned	Matched Query/-ies
1	30	5	Q1
2	15	7	Q2, Q3

Workaround: LLM-Based Generation

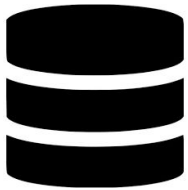
Telemetry Dataset (unknown queries)

CPU Seconds	GB Scanned	Arrival Time
30	5	3
15	3	7

Synthesized Queries
`SELECT * FROM pizzas WHERE name = 'diavola'`
`SELECT COUNT(*) FROM pizzas WHERE type = 'vegetarian'`

Large Language Model

Target System



Example Systems:
SQLBarber(SIGMOD 2026) [4],
ResQ(aRxiV) [5]

Workaround: LLM-Based Generation

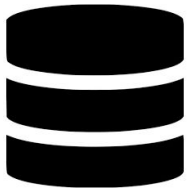
Telemetry Dataset (unknown queries)

CPU Seconds	GB Scanned	Arrival Time
30	5	3
15	3	7

Synthesized Queries
`SELECT * FROM pizzas WHERE name = 'diavola'`
`SELECT COUNT(*) FROM pizzas WHERE type = 'vegetarian'`

Large Language Model

Target System



Example Systems:
SQLBarber(SIGMOD 2026) [4],
ResQ(aRxiV) [5]

Workaround: LLM-Based Generation

Telemetry Dataset (unknown queries)

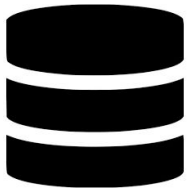
CPU Seconds	GB Scanned	Arrival Time
30	5	3
15	3	7

SQL Pool

Query	CPU Seconds	GB Scanned
Q1	40	4
Q2	5	1

Large Language Model

Target System



Example Systems:
SQLBarber(SIGMOD 2026) [4],
ResQ(aRxiV) [5]

Workaround: LLM-Based Generation

Telemetry Dataset (unknown queries)

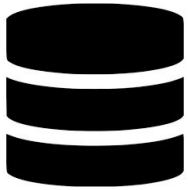
CPU Seconds	GB Scanned	Arrival Time
30	5	3
15	3	7

SQL Pool

Query	CPU Seconds	GB Scanned
Q1	40	4
Q2	5	1

Large Language Model

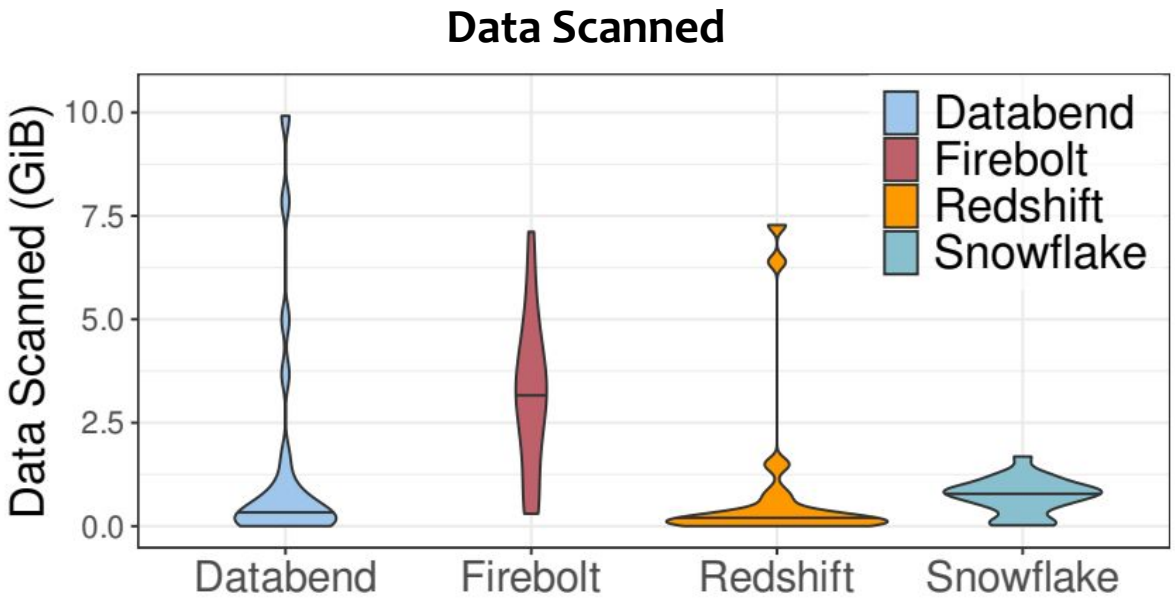
Target System



Example Systems:
SQLBarber(SIGMOD 2026) [4],
ResQ(aRxiV) [5]

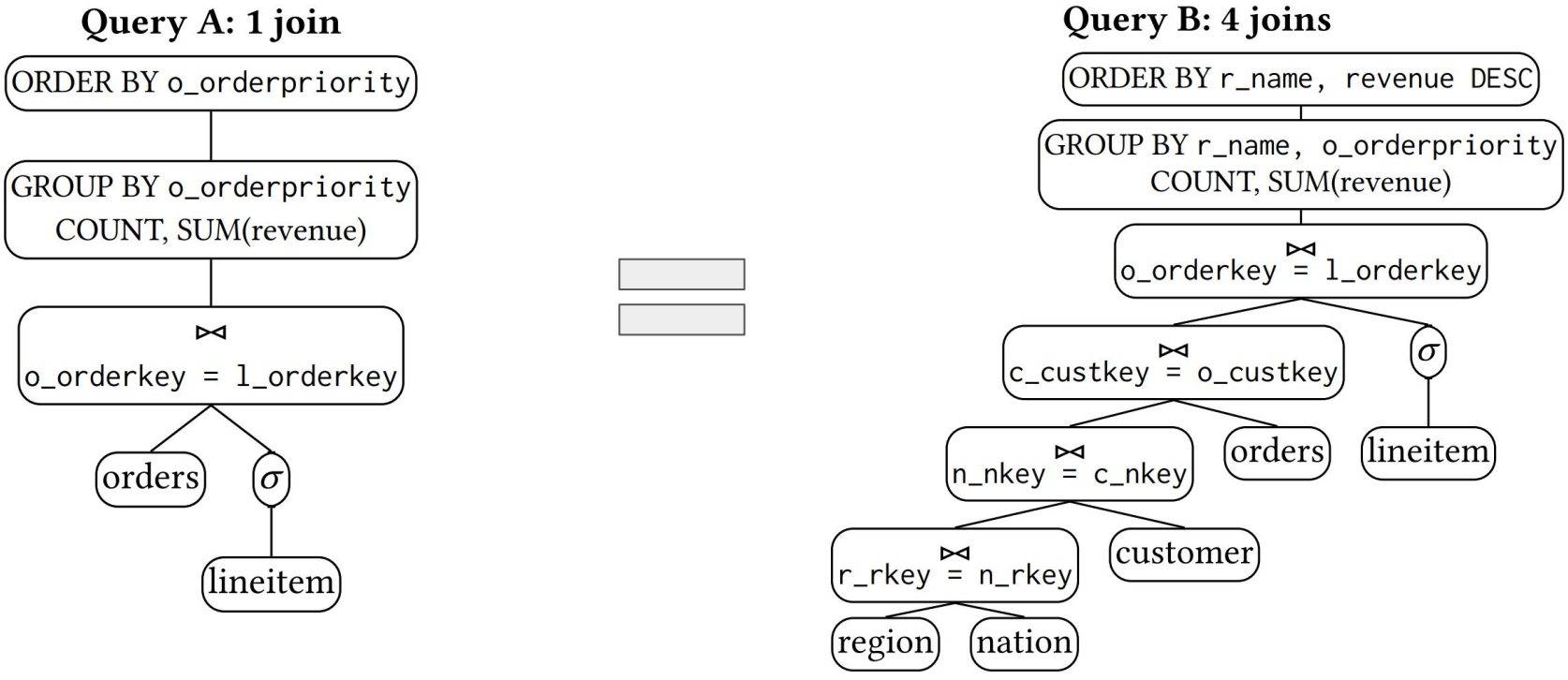
Repeats in a feedback loop until matching succeeds

Experiment 1: Varying the database system



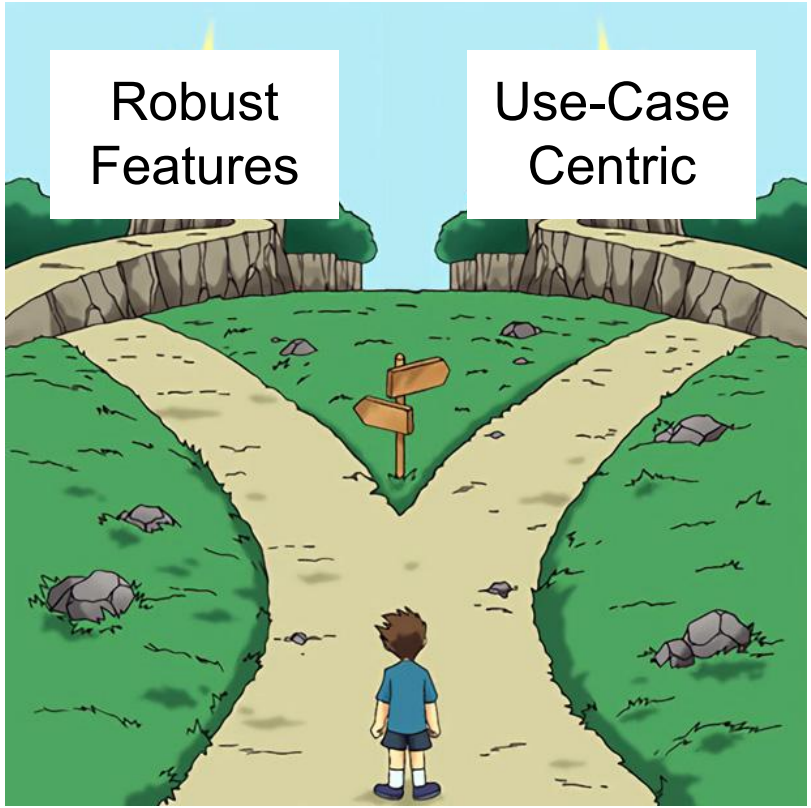
Insight 1: System telemetry describes how a particular engine executed a query

Consequence: No Guarantee on Structural Fidelity



These two queries produce nearly identical telemetry.

Where do we go from here?



Path 1: Transferable Workload Features

Enrich traces with engine-agnostic features beyond telemetry

Query Structure

E.g., subquery depth, CTEs, window functions

Data Characteristics

E.g., null fraction, data-type distribution

Plan-Shape Features

E.g., Join algorithm, parallelism, relative cardinalities

Workload-Level Features

Read/write ratio, access frequency, session type

Path 1: Transferable Workload Features

Enrich traces with engine-agnostic features beyond telemetry

Query Structure

E.g., subquery depth, CTEs, window functions

Data Characteristics

E.g., null fraction, data-type distribution

Plan-Shape Features

E.g., Join algorithm, parallelism, relative cardinalities

Workload-Level Features

Read/write ratio, access frequency, session type

- *Privacy-preserving Query Anonymization: Open Challenge [6]*
- *Recent attempts of standardizing system-agnostic query representations (Substrait [7])*

Example: Our query similarity evaluation

We used 27 AST-based features to measure structural similarity between queries

Count Features (14)

n_tables, n_joins, n_filters,
n_aggregations, n_group_by_cols,
n_order_by_cols, n_select_cols,
n_subqueries, n_and, n_or,
n_comparison_ops, n_math_funcs,
n_string_funcs, n_date_funcs

Boolean Features (12)

has_having, has_limit,
has_distinct, has_union,
has_case, has_window,
has_exists, has_in_subquery,
has_cte, has_between,
has_like, has_not

Structural (1)

depth: maximum
AST nesting depth

Richer query structure information already enabled us to identify weaknesses of generated queries

Path 2: Use-Case-Centric Benchmarks

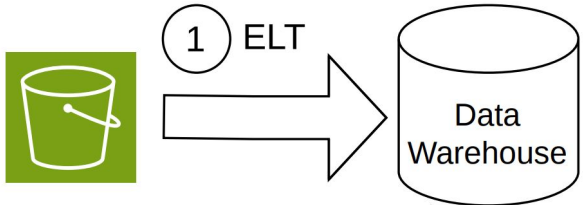
We know how data warehouses are used

- dbt, Superset, Airflow, …
- We also know that machine-generated queries from these tools favor specific SQL patterns (e.g., CTAS)

Instead of reverse-engineering queries, just run the tools.

Path 2: Use-Case-Centric Benchmarks

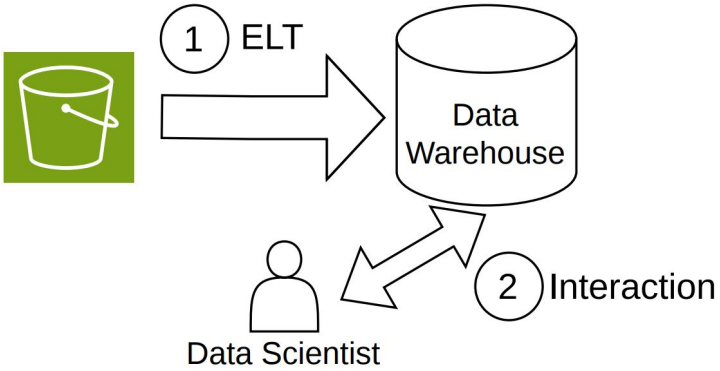
Jaffle Shop e-commerce: dbt (ELT) + Superset (BI) on Firebolt



- Start with a well-defined analysis task

Path 2: Use-Case-Centric Benchmarks

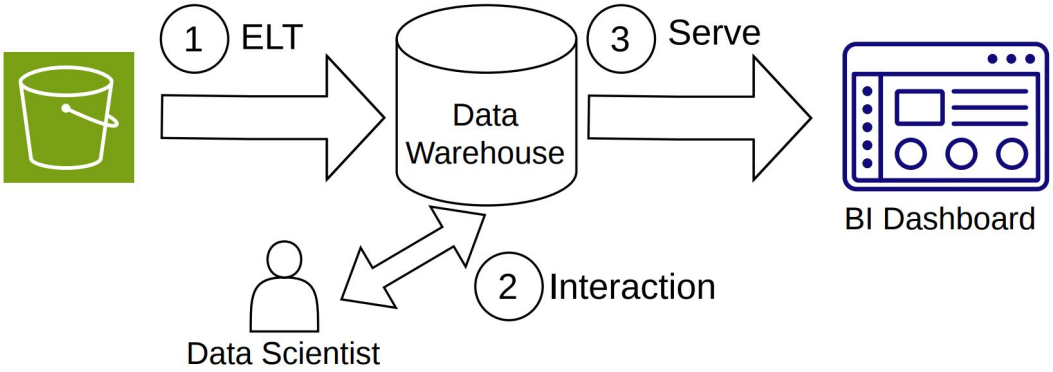
Jaffle Shop e-commerce: dbt (ELT) + Superset (BI) on Firebolt



- Start with a well-defined analysis task
- Identify the relevant business questions

Path 2: Use-Case-Centric Benchmarks

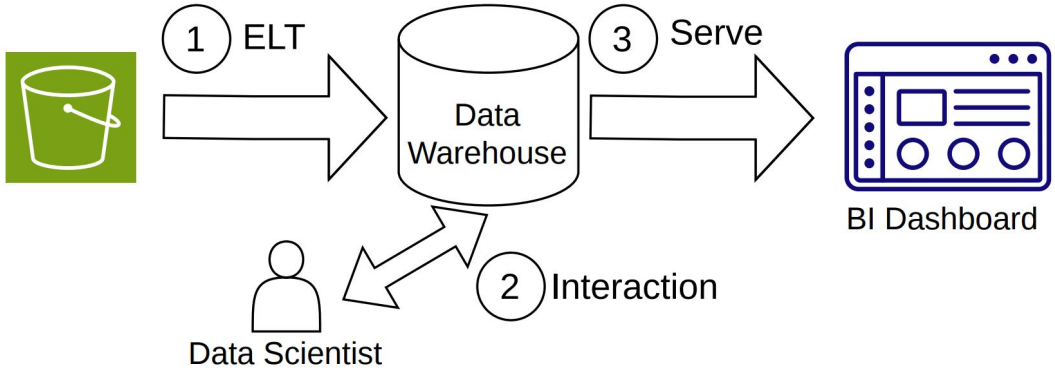
Jaffle Shop e-commerce: dbt (ELT) + Superset (BI) on Firebolt



- Start with a well-defined analysis task
- Identify the relevant business questions
- Let standard data-engineering tools generate the SQL (e.g., dbt, Apache Superset)

Path 2: Use-Case-Centric Benchmarks

Jaffle Shop e-commerce: dbt (ELT) + Superset (BI) on Firebolt



- Start with a well-defined analysis task
- Identify the relevant business questions
- Let standard data-engineering tools generate the SQL (e.g., dbt, Apache Superset)

Queries are transparent by construction

Preliminary Workload Insights

Patterns matching industry traces:

- Joins on string columns
- Timestamp operations
- Structural repetition: same template, different filter values

Patterns absent from synthesis tools:

- CTEs, LIKE, BETWEEN
- dbt compiles its DAG into CTAS sequences (Redset: CTAS accounts for 54.7% of runtimes)
- Real aggregation patterns (no SIN/MD5)

Final Projects

Project Guidelines

Deliverables

- **Code:** implementing the project (Git repo, reproducible).
- **Short report:** summary of findings.
- **Presentation:** ~10 minutes + questions.
- **Deadline:** 14.07

How it works

- 8–9 topics on offer: sign up for one.
- Up to 2 students per topic: two students each own a distinct part, not the same task.
- **Supervisor:** every project has one; check in with them.
- **Duration:** 4-weeks

Project Topic 1: Transferable Workload Features

Telemetry says how one engine ran a query. Can engine-agnostic features describe the query itself portably across systems?

Telemetry (CPU, bytes, op counts) underdetermines a workload — many different query structures hit the same numbers. So a synthetic stream can "match the metrics" yet look nothing like production.

Description: This project builds an engine-agnostic feature extractor and uses it in a loop: generate queries

→ generate a batch of queries

→ run them on the cloud-analytics benchmark; record telemetry and structural features

→ compare that profile against a target (real) workload: the differences show where the synthetic set is off

→ turn those differences into extra generation constraints and regenerate

Deliverables: Feature extractor + workload generator

Relevant Works:

- Cloud Analytics Benchmark
- Redbench

Supervisor: Michalis Georgoulakis (michalis.georgoulakis@tum.de)

Project Topic 2: Use-Case-Centric Benchmarks

Instead of reverse-engineering SQL from telemetry, start from a real application and let standard tools generate the workload.

Description: Synthetic benchmarks (TPC-H/DS) are shaped by a designer; real traces are private. This project takes a third path — build a workload from a real use case with real tools, so the SQL emerges from the stack instead of being hand-written:

→ pick an application scenario and its business questions

→ let dbt (ELT) and Superset (BI) generate the actual SQL — queries the tools emit, not a benchmark author

→ capture the resulting workload (the real SQL the stack runs)

→ characterize its patterns against synthetic benchmarks (TPC-H/DS) and industry traces (Snowset/Redset)

Deliver: A runnable use-case workload + a report contrasting its query patterns with a synthetic baseline.

Relevant works:

- Workload Insights from Snowflake
- Why TPC is not Enough: An Analysis of the Redshift Fleet

Supervisor: Michalis Georgoulakis (michalis.georgoulakis@tum.de)

Project Topic 3: Statistics for Object-Store Pruning

On object storage, request count is the currency. Which statistics let a reader skip data — and when does the metadata they cost outweigh what they save?

Given: Parquet data (optionally Delta / Iceberg tables) on Azure Blob; an engine that exposes pruning (e.g., DuckDB); selective queries spanning a range of selectivities.

Core task: Quantify, in object-store terms, how effective each pruning statistic is — partition pruning, row-group min/max (zone maps), the Parquet page index, and bloom filters. For each, report data skipped, object-store requests (metadata vs data), read amplification, and latency as a function of selectivity.

- Fine-grained stats trade extra metadata reads for skipping - find where each pays off.
- More metadata trade extra storage space for skipping.

Deliver: A report comparing pruning effectiveness per statistic across selectivities.

Resources:

- Pruning in Snowflake: Working smarter, not harder
- When metadata is big data

Supervisor: Michalis Georgoulakis (michalis.georgoulakis@tum.de)

Project Topic 4: Provisioned + Serverless Scheduling

Provisioned pools cost money while idle; serverless pay-per-query gets pricey under load. Can a scheduler route each query to the cheaper one without missing a latency target?

Given: Azure Synapse: a provisioned (dedicated) SQL pool (pay-per-hour, fixed capacity) and the serverless SQL pool (pay-per-TB-scanned, scales to zero) and an input workload with varying arrival patterns.

Core task: Build a scheduler that places each incoming query on the provisioned pool or on serverless SQL: based on its size and the current load to minimize cost subject to a latency target, or the opposite.

Deliver: Working scheduler and performance numbers

Bonus: Pause / resume or resize the provisioned pool to exploit idle windows.

Resources:

- Saving Money for Analytical Workloads in the Cloud

Supervisor: Michalis Georgoulakis (michalis.georgoulakis@tum.de)

Project Topic 5: Distributed DuckDB via Quack

How much of a hand-written distributed engine can a declarative layer recover? We “hack” Quack’s remote-scan capability to mimic distributed execution.

Given: DuckDB (a fast single-node, in-process engine) and Quack — one DuckDB issues remote scans against others with predicate pushdown. Note: Quack targets client-server-style access, not distribution; we repurpose it.

Goal: Hash-partition a relation across K DuckDB instances purely in SQL (the shuffle as remote scans + pushdown) and demonstrate a distributed aggregation. Characterize: scaling with K, bytes actually shuffled (and how much pushdown reduces them), and end-to-end latency vs a single DuckDB. Identify where the declarative path leaves performance on the table.

Optional: Extend to a distributed hash join by co-partitioning both inputs.

Resources:

- **Quack:** The DuckDB Remote Client-Server Protocol

Supervisor: Maximilian Rieger (riegerm@in.tum.de)

Broadcast vs. Partitioned Joins

An empirical study of when each strategy wins

The core question

A broadcast join replicates the build side to every worker; a partitioned (shuffle) join re-partitions **both** inputs by the join key. Which should the optimizer pick — and where exactly does the trade-off flip?

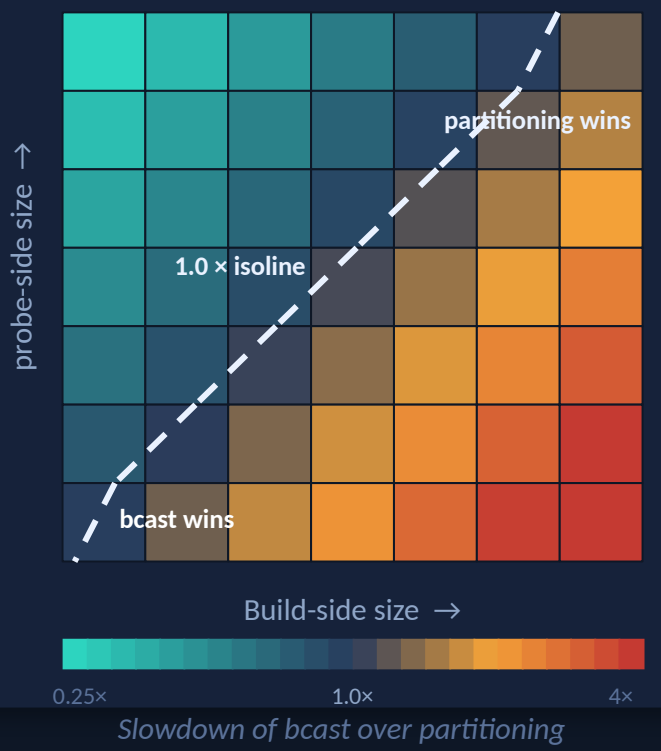
Build side drives cost

Conventional wisdom: the deciding factor is the size of the build-side hash table that must be broadcast.

Probe side is never shuffled

Broadcast's hidden edge — the (often huge) probe side stays put, avoiding a full re-partition over the network.

2D slowdown heatmap



Slowdown of bcast over partitioning

WHAT TO LOOK FOR

01

Where does the 1.0× isoline actually fall?

02

How far does a large probe side push the boundary toward bigger builds?

03

Two base table filters let you sweep the size space in 2D.

Contact: Abdelrahman Adel (abdelhaa@cit.tum.de)

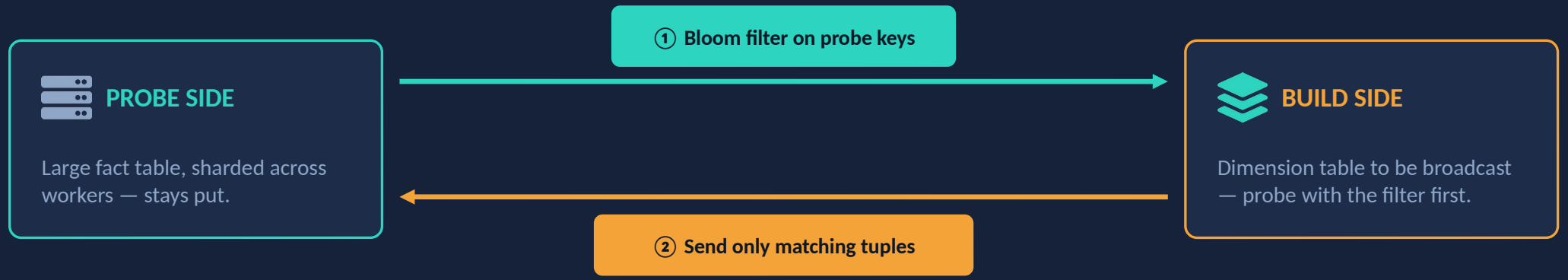
4-WEEK PROJECT PHASE

PROJECT 02

Semi-Join Reduction with Bloom Filters

Shrink the broadcast by filtering before you send

THE REDUCTION PIPELINE



QUESTION A

How big should the filter be?

Bits per key trades memory and network for a lower false-positive rate. Too small and junk tuples survive; too large and you erode the savings.

- Sweep bits/key vs. tuples sent
- Find the false-positive sweet spot
- When is it worth an extra round-trip (bloom filter on build-side AND probe side)?

QUESTION B

Per-worker or one combined filter?

Per-worker filters are cheap but each carries its own false positives. One merged (OR-combined) filter is tighter — but re-scanning the build side per worker costs passes.

- Per-worker FPR vs. combined filter
- Extra build-side scans/checks worth it?

Contact: Abdelrahman Adel (abdelhaa@cit.tum.de)

Project Topic 8: Open Table Formats on Object Storage

How do Delta Lake and Iceberg actually use object storage underneath a workload: and how have they changed since 2023?

Given: Spark + Delta Lake & Apache Iceberg (Hudi optional) on Azure Blob; Jain et al. (CIDR 2023) as the methodological starting point.

Core task: Bring Spark to current Delta / Iceberg releases and reproduce ONE workload class: point lookups, merge/upsert, or time-travel. Measure object-store behavior underneath: number & size of requests, metadata read amplification, and latency per format.

Deliver: (a) a results report for the chosen workload across both formats; (b) a current feature-and-version matrix showing how the formats evolved since 2023

Bonus: Add Apache Hudi to the comparison, and/or extend to a second workload class.

Supervisor: Tobias Götz (goetzt@in.tum.de)

Project Topic 9: Object-Store-Native Transactions

Can transactional reads be served purely through object-store semantics and no external coordinator? Characterize the request pattern that buys.

Given: The LakeVilla (LV) prototype (C++) with its existing Delta Lake path; hudi-rs (Hudi's Rust implementation) and its C++ bindings; Azure Blob Storage.

Core task: Characterize existing Azure Blob data path wrt throughput, latency, and the object-store request pattern LV generates to provide transactional reads with no external coordinator. Implement LV[Read] for Apache Hudi via the Rust C++ bindings (a Copy-on-Write snapshot read suffices) and compare its object-store behavior against the Delta baseline.

Bonus: LV[Write].

Resources:

- LakeVilla

Supervisor: Tobias Götz (goetzt@in.tum.de)

Next Week

- **Topic Selection Confirmation:** Everyone tells us what they plan to work on.

- **Praktikum Outlook:** Hardware trends in the cloud and open research directions

- **Lab 6:** Oral Discussion
  - Deadline: **18.06, 23:59:59**