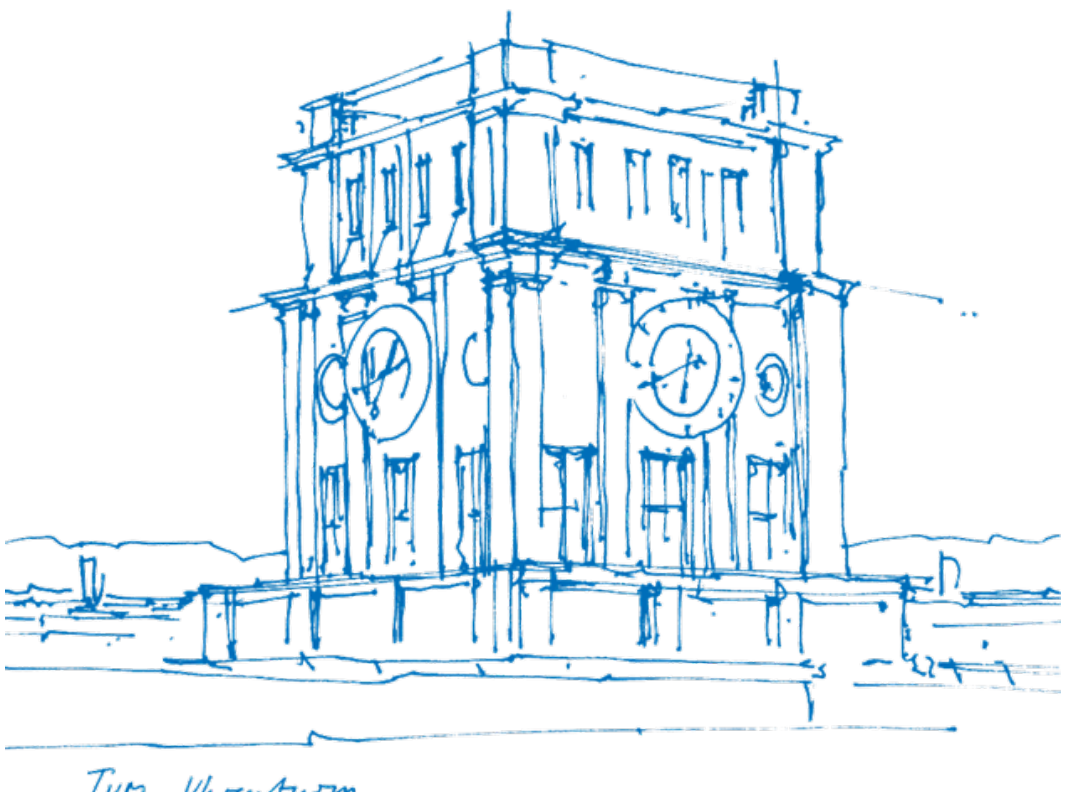
School of Computation, Information and Technology
Technical University of Munich

DataPro 2026

Projects



Mateusz Gienieczko, Mykola Morozov

School of Computation, Information and Technology
Technical University of Munich

2026.06.10

Projects!

You have until **15.07.2026** to complete.

The scope is guided by the advisor. We expect a short (< 10 minutes) presentation at the end.

Rapid Bloom Filters with Compact LUTs

Bloom filters are fast and compact probabilistic data structures that allow for set-membership queries. They are shaped like a bitset. Queries read and check multiple bits which can become a performance bottleneck with increasing size. Apache Arrow solves this with a trick involving a compact lookup table of bit configurations, allowing checking many bits with few instructions.

Rapid Bloom Filters with Compact LUTs

Supervisor: Altan Birler (altan.birler@tum.de)

Goals

- Implement bloom filter with compact mask lookup tables and variations.
- Make it fast/faster/fastest.
- Measure both performance and false-positive rate of the variations.
- Can you make the trick work well for 16-bit sized register-blocked bloom filters as well?

Rapid Bloom Filters with Compact LUTs

Resources:

- https://github.com/apache/arrow/blob/main/cpp/src/arrow/acero/bloom_filter.h

Efficient Automata for CSV Parsing

We have built a very fast (>100 GB/s) parallel, vectorized CSV parser. It works on common cases, but for completeness we have a DFA fallback. The DFA works and is reasonably fast, but it was written pragmatically and has not been studied on its own.

Efficient Automata for CSV Parsing

Supervisor: Simon Ellmann (ellmann@in.tum.de)

Goals

- Survey existing parsers (Rust csv, DuckDB).
- Benchmark existing solutions under different workloads.
- What is the hot path/bottleneck?
- Is there a design that looks like a good default?

Efficient Automata for CSV Parsing

Resources:

- https://github.com/BurntSushi/rust-csv
- Raasveldt, Mühleisen. DuckDB: an Embeddable Analytical Database. SIGMOD ‘19.

NUMA-Aware Concurrent Graph Data Structures

Sortledton, published by our group at VLDB 2022, is a universal graph data structure. Its concurrency uses one read-write latch per vertex — simple and effective, but with two known limitations: contention under bursty workloads, and no NUMA awareness. On multi-socket machines, vertex data may reside on a remote NUMA node from the accessing thread, incurring penalties that grow with socket count

NUMA-Aware Concurrent Graph Data Structures

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

Goals

- Investigate Sortledton's scalability on a multi-socket machine (access will be given).
- From that evidence, pick a bottleneck and optimize it along at least one axis – cache awareness, vectorization (SIMD), or synchronization.

NUMA-Aware Concurrent Graph Data Structures

Resources:

- https://gitlab.db.in.tum.de/mike_georg/graph-data-structure-student-topic
- P. Fuchs, D. Margan, J. Giceva. Sortledton: a Universal, Transactional Graph Data Structure. PVLDB 15(6), 2022.

High-Performance GPU Code for Text Parsing

Text parsing with Deterministic Finite Automata (DFAs) is the backbone of pattern matching, but making it fast on modern hardware is notoriously hard. Processing the text character-by-character and jumping all over memory is the bottleneck of the current state-of-the-art aproaches. To break this bottleneck, we need to process multiple characters at the exact same time. We've already experimented with generating SIMD (SSE4.2) instructions for the CPU with mixed results. Now, we want to see what happens when we unleash the massive parallelism of the GPU.

High-Performance GPU Code for Text Parsing

Supervisor: Calin-George Pop (calin.george.pop@tum.de)

Goals

- Implement a GPU-based kernel and compare against the existing code generation for LLVM (benchmarks already exist, you will have to analyse results).
- Implement GPU-based code generator (targeting C++/CUDA or LLVM if you are brave).

High-Performance GPU Code for Text Parsing

Resources:

- https://github.com/calin2110/FSST-LIKE-Matching

Adaptive Query Execution Based on Hardware Occupancy

Executing queries with GPU acceleration often runs into bottlenecks since GPUs require upfront allocation of resources. A modern query executor needs to balance resources and schedule queries appropriately.

Adaptive Query Execution Based on Hardware Occupancy

Supervisor: Mykola Morozov (mykola.morozov@tum.de)

Goals

- Implement a “dumb” baseline (always schedule on CPU, always schedule on GPU with fixed resources).
- Find improvements step by step – schedule on the GPU if resources permit; allocate resources based on size estimates.
- Get creative (or have Mykola get creative for you)!

Flying Start for Query Execution on GPUs

Compiling and optimising GPU code takes a long time. In case of CPU execution the latency incurred by compilation has been successfully studied and solved with Flying Start. The query is compiled in the background while the engine starts executing the unoptimised version quickly. Could a similar scheme work for GPUs.

Flying Start for Query Execution on GPUs

Supervisor: Mykola Morozov (mykola.morozov@tum.de)

Goals

- Implement a solution that starts executing a query on the CPU while GPU is compiled and then switches.
- Investigate at what checkpoints switching makes sense and is efficient.
- Measure against both baselines (full CPU execution, full GPU execution).

Flying Start for Query Execution on GPUs

Resources:

- T. Kersten, V. Leis, T. Neumann: Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. The VLDB Journal, Volume 30, Issue 5

SIMD-accelerated Depth-Aware Subtree String Search

Modern JSON query engines use streaming and subtree skipping to achieve maximum performance. Two dedicated SIMD pipelines form the backbone of `rsonpath`, the state-of-the-art engine: string search (`memmem`) and depth-skipping. One finds keys equal to one in the query, the other skips a JSON subtree by tracking depth and matching brackets.

Sometimes, we know that we want to skip *to a key within the current subtree*. Since SIMD is not easily composable, a dedicated pipeline that fuses the two is required.

SIMD-accelerated Depth-Aware Subtree String Search

Supervisor: Mateusz Gienieczko (giem@in.tum.de)

Goals

- Investigate the SIMD pipelines for string search and depth-skipping.
- Write a fuzed pipeline using AVX2 and benchmark it on hardcoded queries.
- Bonus: put this into the full engine.

SIMD-accelerated Depth-Aware Subtree String Search

Resources:

- M. Gienieczko, F. Murlak, C. Paperman: Supporting Descendants in SIMD-Accelerated JSONPath, ASPLOS'23
- https://github.com/rsonquery/rsonpath

Rapid JSON Compression and Prettifier

Supervisor: Mateusz Gienieczko (giem@in.tum.de)

Goals

- Implement a SIMD prettifier (seems easier).
- Compare to existing tools (e.g. `jq`).
- Implement a SIMD compressor using AVX512.
- Compare to existing tools (e.g. `jq -c`).

Rapid JSON Compression and Prettifier

Resources:

- M. Gienieczko, F. Murlak, C. Paperman: Supporting Descendants in SIMD-Accelerated JSONPath, ASPLOS'23
- D. Lemire, G. Langdale: Parsing Gigabytes of JSON per Second, VLDB Journal 28 (6)
- https://github.com/rsonquery/rsonpath
- https://github.com/simdjson/simdjson