How building an industry DBMS differs from building a research one

Markus Dreseler @ CIDR

January 2023 - PUBLIC

Background

- 2015 - 2021: PhD student at the chair of Prof. Plattner @ **HPI** Potsdam
  - Built **Hyrise**, an open-source in-memory research DBMS*
- Since April 2021: Senior Software Engineer at **Snowflake** in our Berlin office

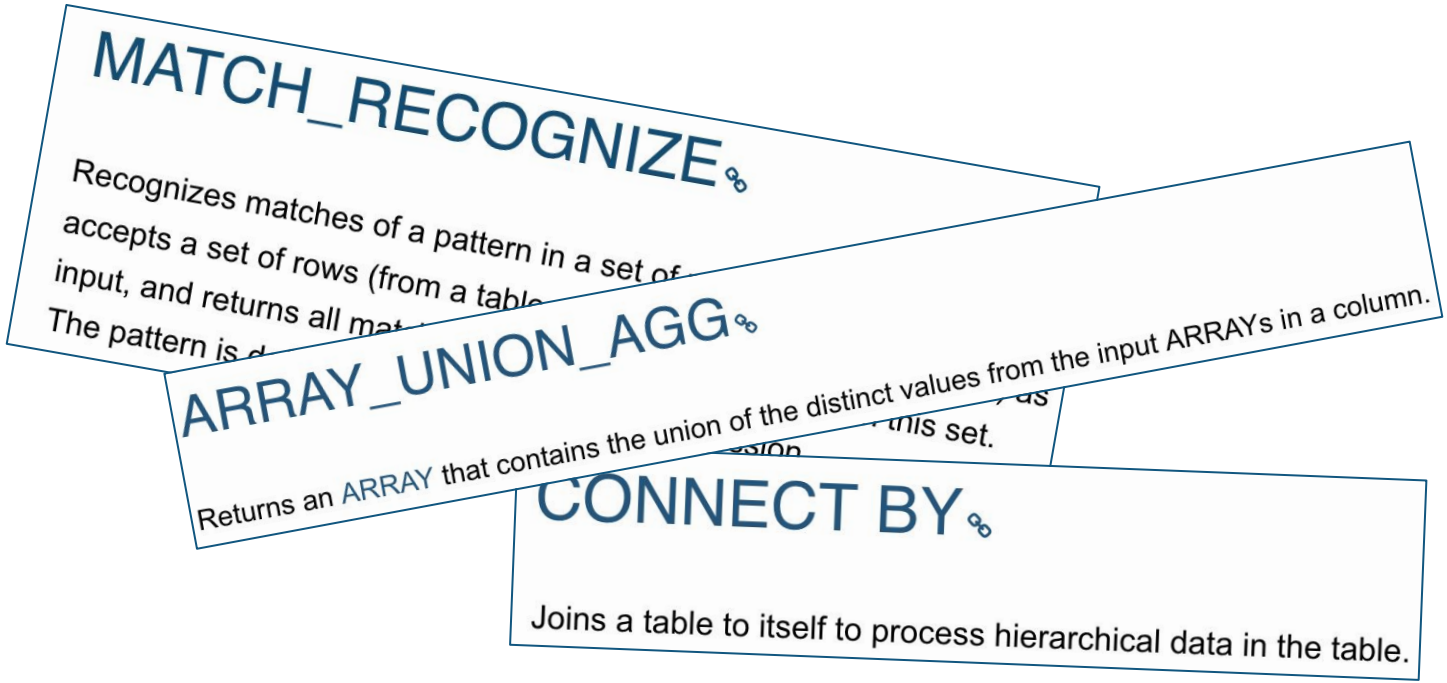
- This talk is based on my personal experience
- It is neither representative nor comprehensive

*) Dreseler et al., Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management (EDBT 2019) - https://github.com/hyrise/hyrise

Snowflake builds a DBMS, right?

- When I was looking for jobs, I thought that Snowflake built a cloud DBMS

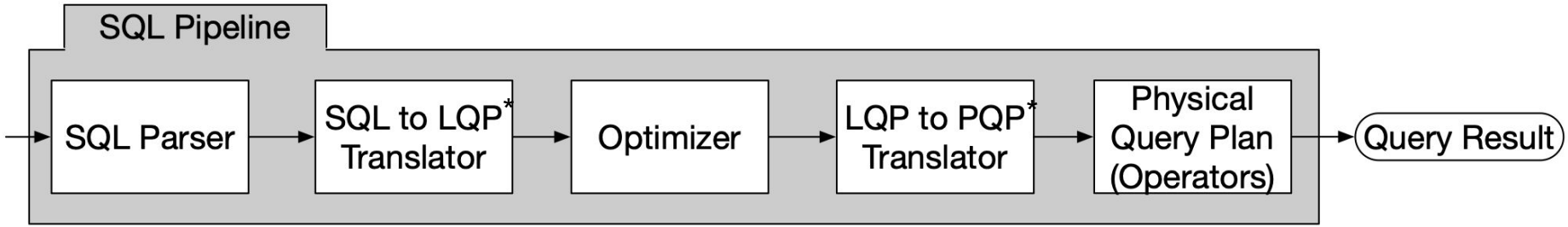
But at least I know SQL…



Similarities

Databases really work like they teach in university

Everything is more complex in an industry database, but the SQL core is similar:



*) Logical / Physical Query Plan

Still doing R&D and still writing papers

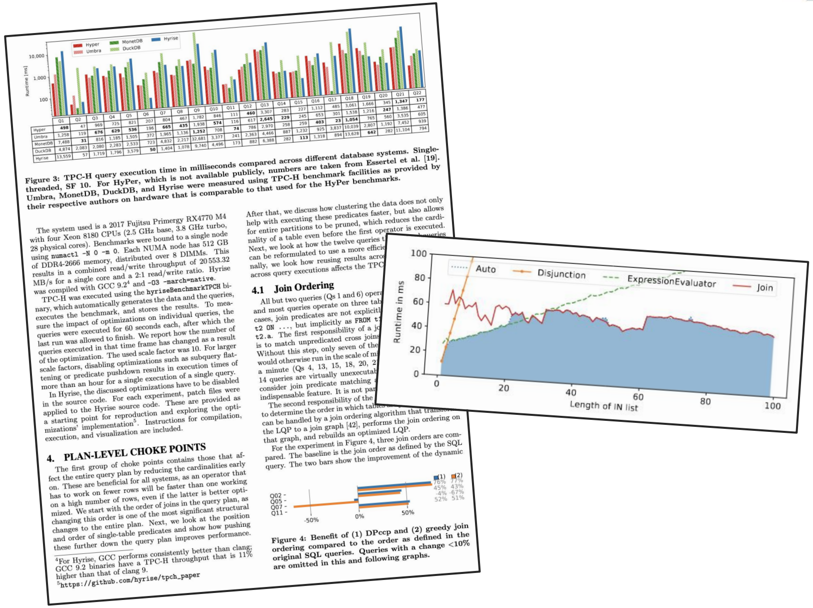


Figure 3: TPC-H query execution time in milliseconds compared across different database systems. Single-threaded, SF 10. For HyPer, which is not available publicly, numbers are taken from Essertel et al. [19]. Umbra, MonetDB, DuckDB, and Hyrise were measured using TPC-H benchmark facilities as provided by their respective authors on hardware that is comparable to that used for the HyPer benchmarks.

The system used is a 2017 Fujitsu Primergy RX4770 M4 with four Xeon 8180 CPUs (2.5 GHz base, 3.8 GHz turbo, 28 physical cores). Benchmarks were bound to a single node using numactl -N 0 -m 0. Each NUMA node has 512 GB of DDR4-2666 memory, distributed over 8 DIMMs. This results in a combined read/write throughput of 20553.32 MB/s for a single core and a 2:1 read/write ratio. Hyrise was compiled with GCC 9.2[4] and -O3 -march=native.

TPC-H was executed using the hyriseBenchmarkTPCH binary, which automatically generates the data and the queries, executes the benchmark, and stores the results. To measure the impact of optimizations on individual queries, the queries were executed for 60 seconds each, after which the last run was allowed to finish. We report how the number of queries executed in that time frame has changed as a result of the optimization. The used scale factor was 10. For larger scale factors, disabling optimizations such as subquery flattening or predicate pushdown results in execution times of more than an hour for a single execution of a single query.

In Hyrise, the discussed optimizations have to be disabled in the source code. For each experiment, patch files were applied to the Hyrise source code. These are provided as a starting point for reproduction and exploring the optimizations' implementation[5]. Instructions for compilation, execution, and visualization are included.

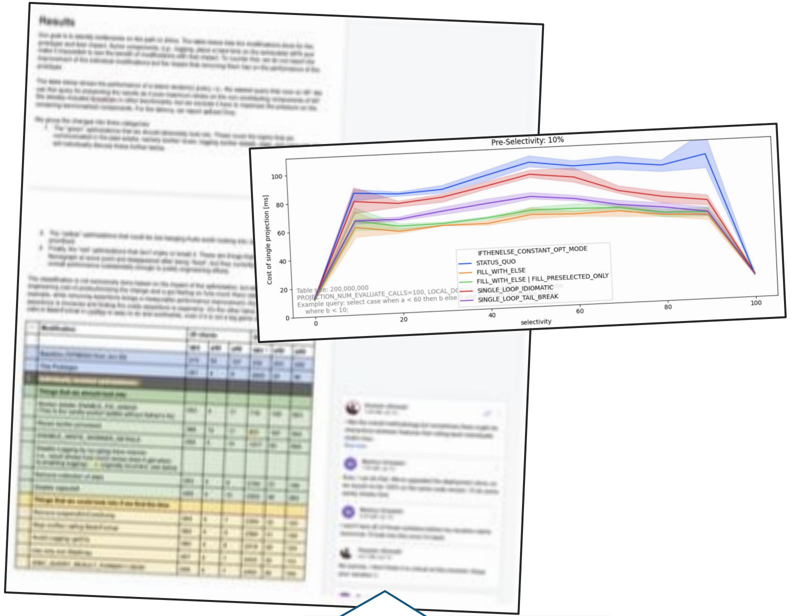
4. PLAN-LEVEL CHOKE POINTS

4.1 Join Ordering

Figure 4: Benefit of (1) DPccp and (2) greedy join ordering compared to the order as defined in the original SQL queries. Queries with a change <10% are omitted in this and following graphs.

[4] For Hyrise, GCC performs consistently better than clang; GCC 9.2 binaries have a TPC-H throughput that is 11% higher than that of clang 9.

[5] https://github.com/hyrise/tpch_paper

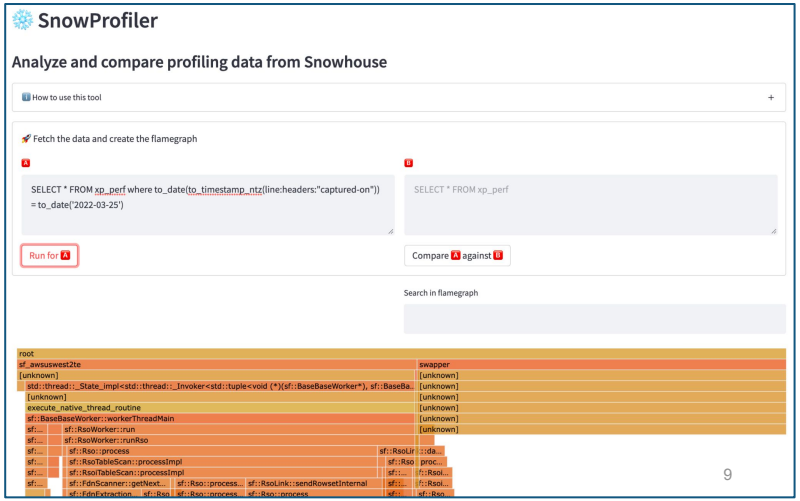


Instant feedback and discussions

Workloads and Optimizations

Relevance of optimizations

- In research, incremental improvements rarely cause much excitement
- In industry, if we make every query 1ms faster, we save a year worth of compute a day
  - Optimizations do not have to be fancy
- **Finding the right thing** to work on
  - Customer requests
  - 15+ PB of metadata in Snowhouse
  - Background tooling across the entire fleet
- But: Having 5% of queries regress is only fine in research



Reliability

Testing and Validation

Unit Tests
End-to-End Regression Tests
Sanitizer Builds
Static Code Analysis — We had this in Hyrise

Query Permutation Testing — This was new for me
Re-executing production queries*
Up- and downgrade testing

*) Yan et al., Snowtrail: Testing with Production Queries on a Cloud Database, DBTest '18

Parameter protection (aka. feature flags)

- Used to **guard new code paths**:

```
603       -   m_missSel->fillWithSubtractedVectors(*probeSel, *selection);
      609 +   if (InitParams::getParamBool(PRM_ENABLE_FIX_360233)) {
      610 +     m_missSel->fillWithSubtractedSubsequenceVectors(*probeSel, *selection);
      611 +   } else {
      612 +     m_missSel->fillWithSubtractedVectors(*probeSel, *selection);
      613 +   }
```

- Multi-level - can be enabled for individual queries
- This allows us to have a **single binary** while still supporting
  - Running code in test environments first
  - Private and public previews
  - Behavior changes

Edge Cases and Resilience

- At **2,400,000,000 queries a day**, everything you can think of happens
  - Race conditions - if it is in the code, it will happen
  - ECC RAM failing / Bit Rot
  - Cloud providers being out of instances
  - Cloud providers giving us bad instances

A researcher could just restart the experiment

Wrapping up

What do I miss?

- Being able to know each part of the code
- Hacking up a performance improvement between a lecture and lunch
- Not worrying about regressions

What is great?

- Code is used billion times a day
- Data-driven development (Snowhouse)
- Support Rotation is stressful, but allows you to have an immediate impact on blocked customers
- Working alongside hundreds of engineers with different expertise

How can you be part of this?

Try Snowflake

You can try Snowflake for 30 days and spend $400 on us:

https://signup.snowflake.com/

Chat with us at CIDR


Marcin Zukowski

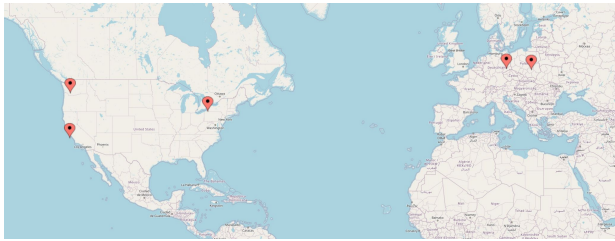

Berni Schiefer


Florian Funke


Jan Kossmann


Markus Dreseler

Join us



- Engineering offices in Berlin, San Mateo, Warsaw, Toronto, Bellevue