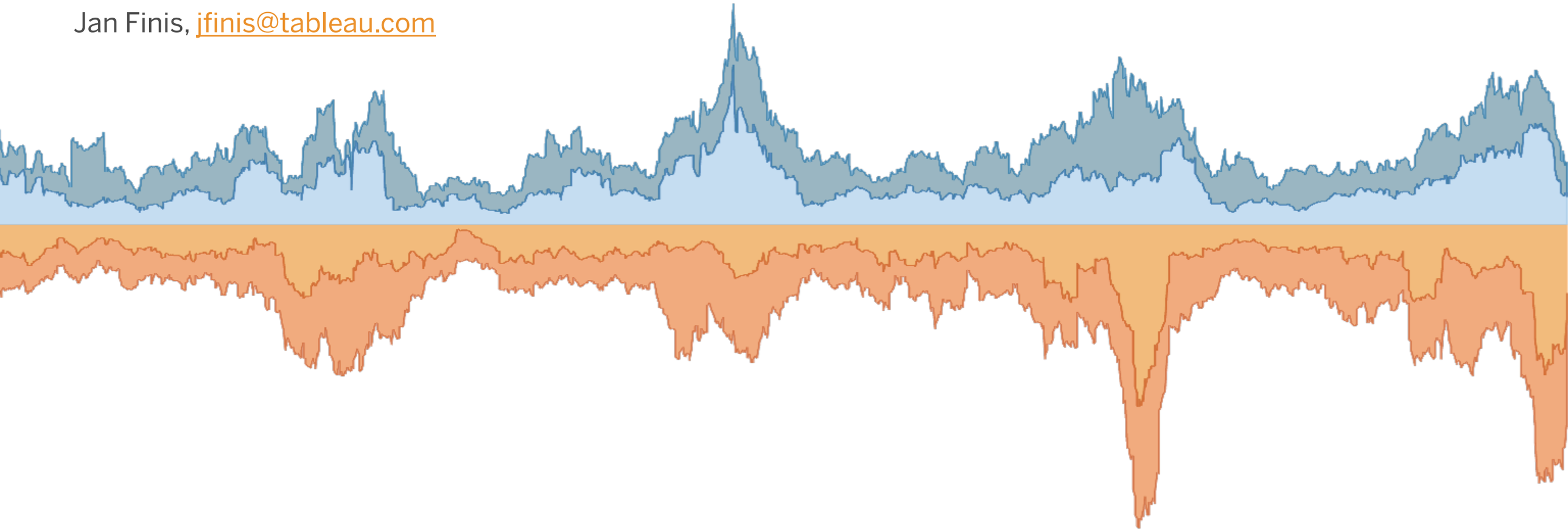
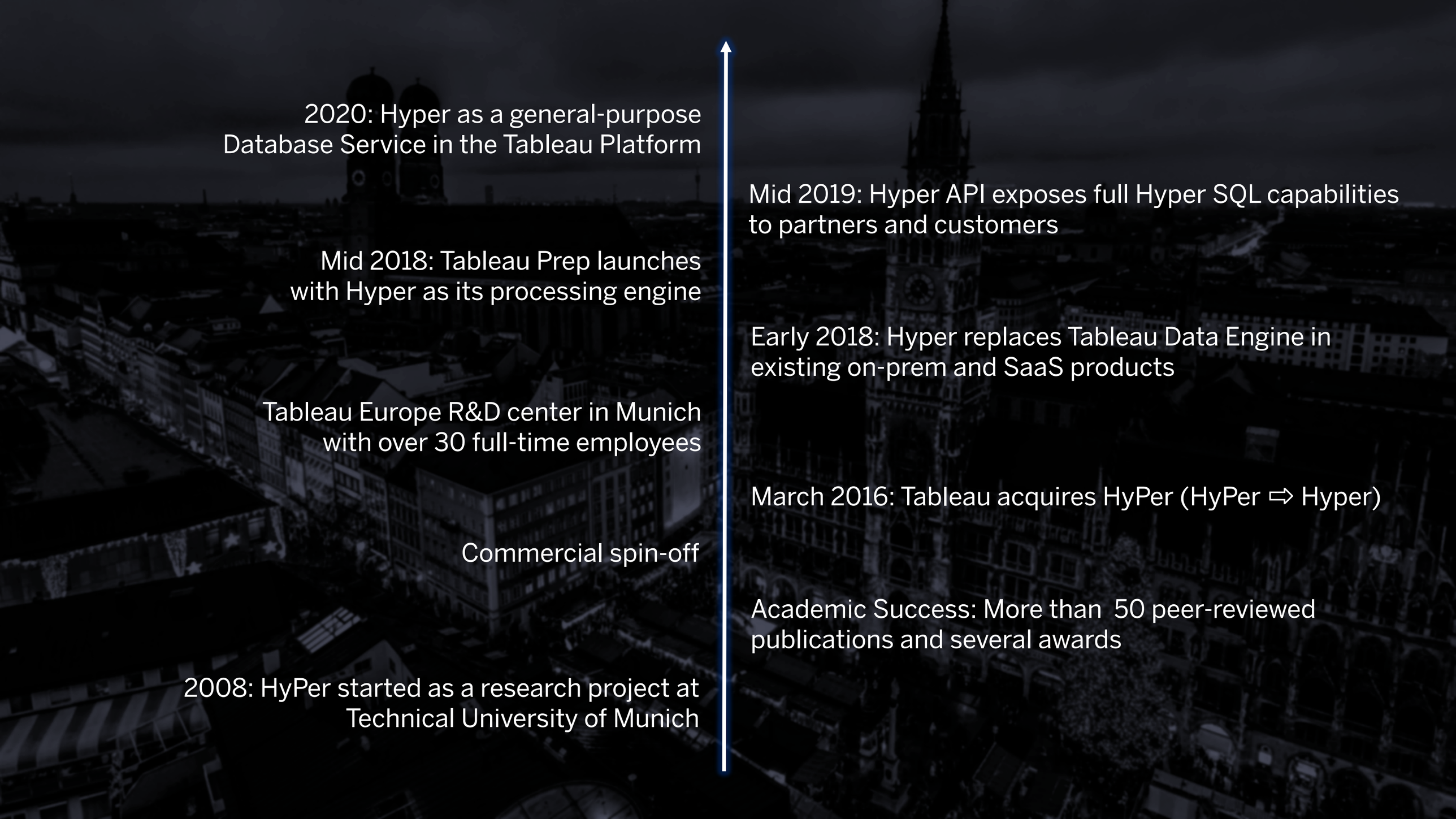
From HyPer to Hyper

Integrating an academic DBMS into a leading analytics and business intelligence platform

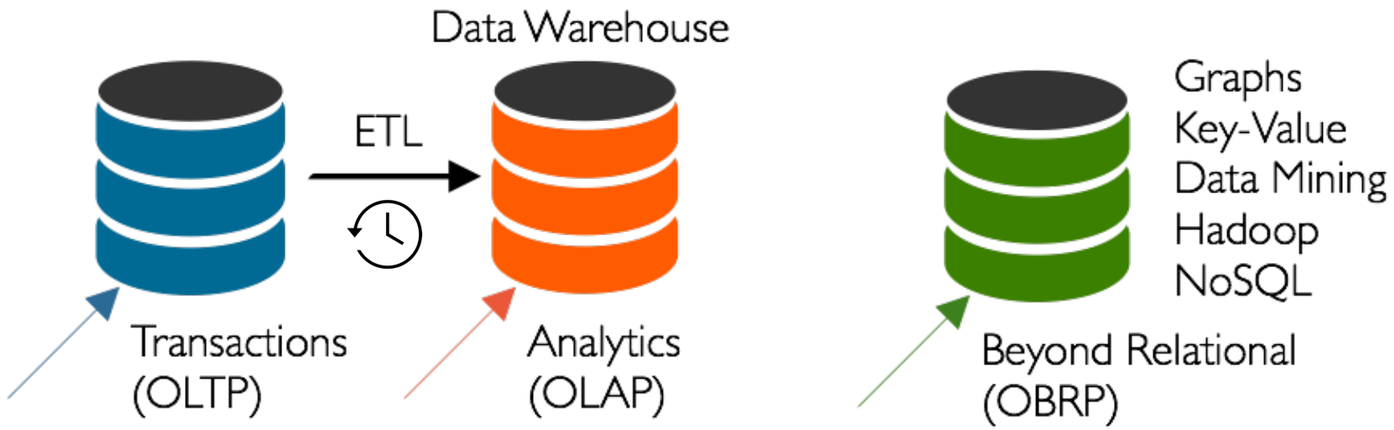
Tobias Muehlbauer, tmuehlbauer@tableau.com
Jan Finis, jfinis@tableau.com



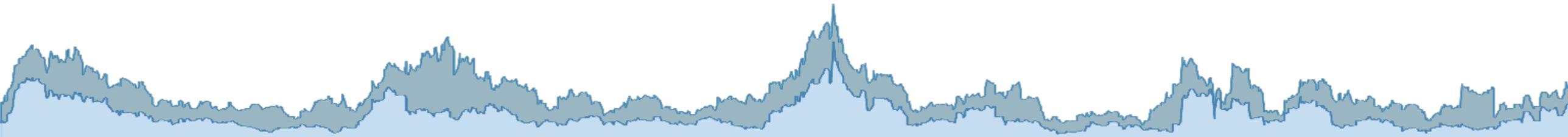
The Story of Hyper


2020: Hyper as a general-purpose Database Service in the Tableau Platform
Mid 2019: Hyper API exposes full Hyper SQL capabilities to partners and customers
Mid 2018: Tableau Prep launches with Hyper as its processing engine
Early 2018: Hyper replaces Tableau Data Engine in existing on-prem and SaaS products
Tableau Europe R&D center in Munich with over 30 full-time employees
March 2016: Tableau acquires HyPer (HyPer ⇨ Hyper)
Commercial spin-off
Academic Success: More than 50 peer-reviewed publications and several awards
2008: HyPer started as a research project at Technical University of Munich

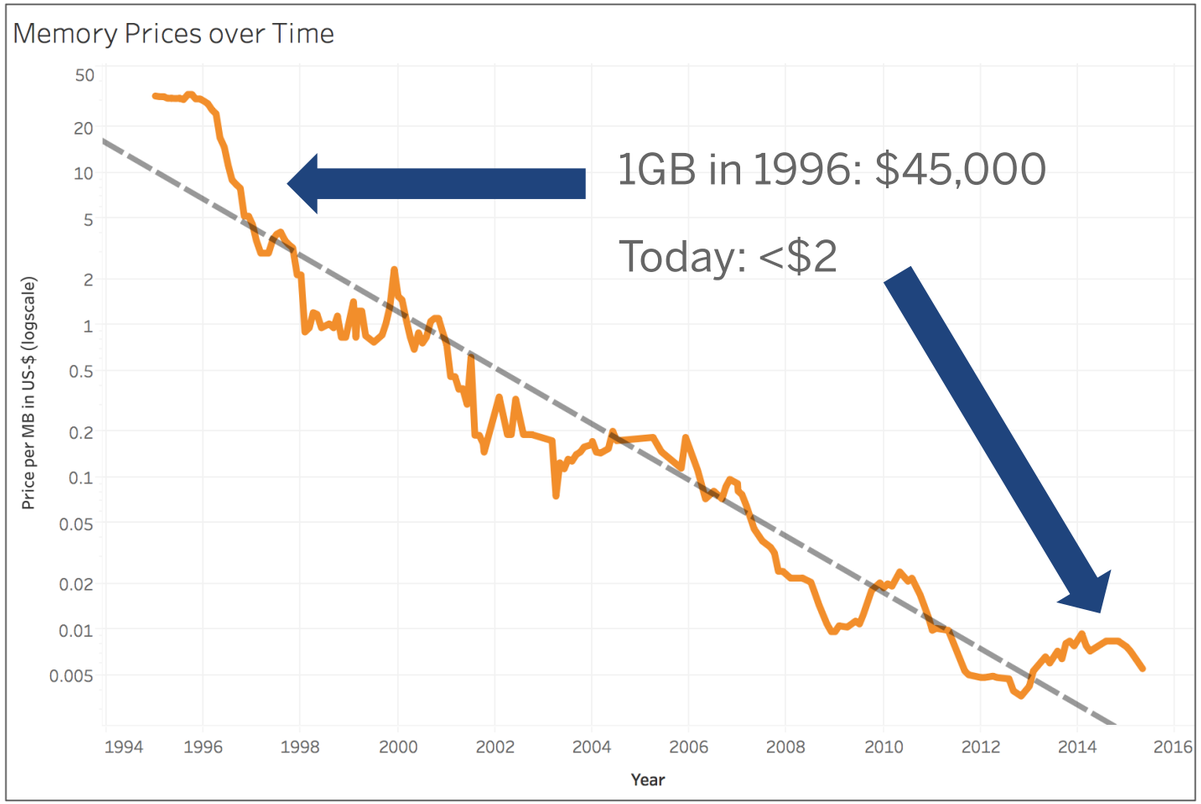
Context: One Size Fits All?

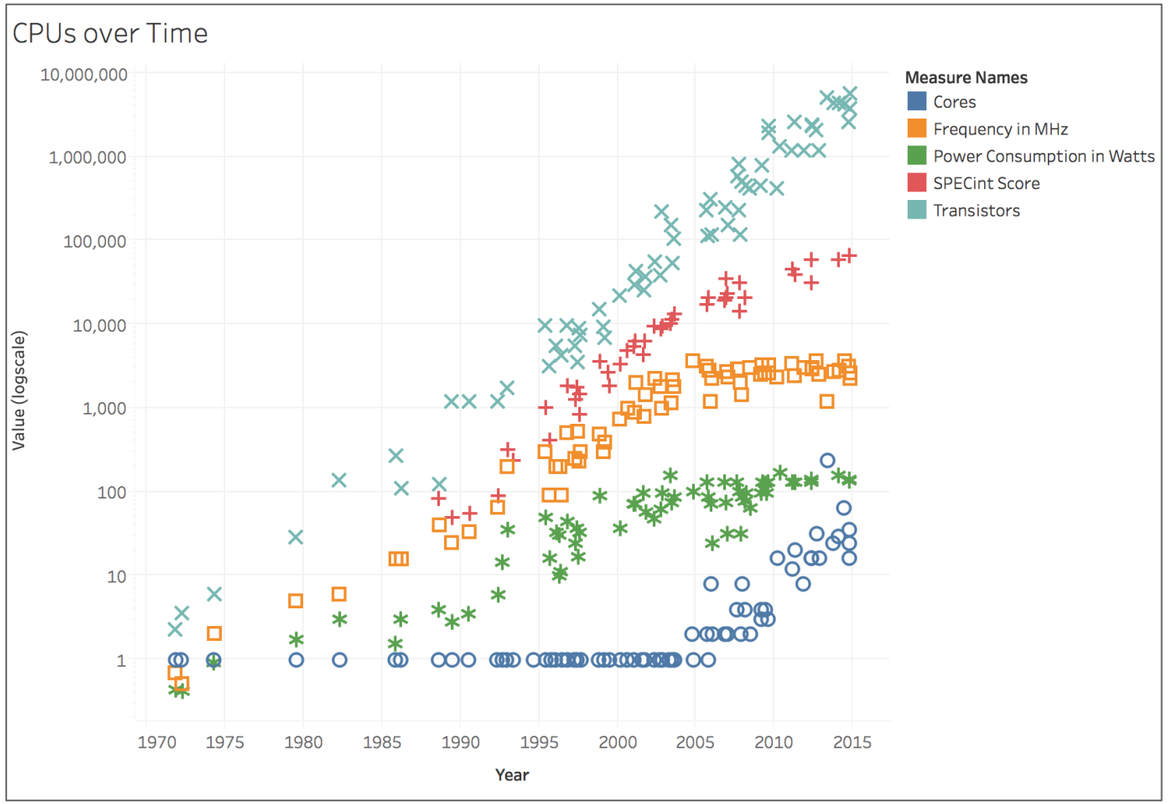


One size does **not** fit all, but what about data lag and disconnect?

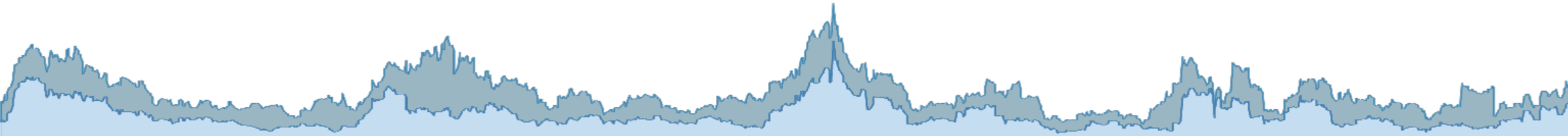


Context: A Changing Hardware Landscape





In order to leverage modern hardware, **databases need to change**.

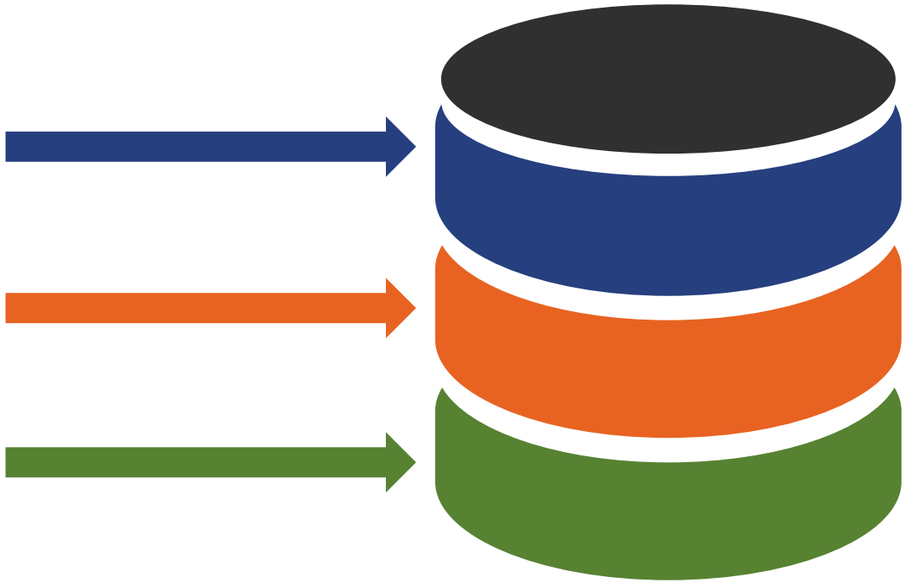


The Idea Behind Hyper

Operational
Data Ingestion, Transactions, and Streaming

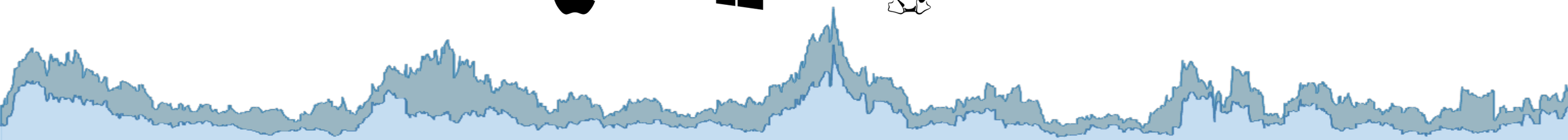
Analytical
On Latest State of the Data

Beyond Relational



*Prototype

Optimized for Underlying Hardware & Operating System



One System

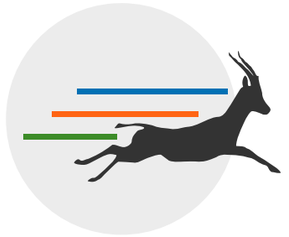
Hyper is developed as a **general-purpose database system** that combines **transaction processing, data ingestion, and data analytics.**

One State

Transaction processing, data ingestion, and data warehousing all on the same state to **enable real-time analytics of the latest state of your data.**

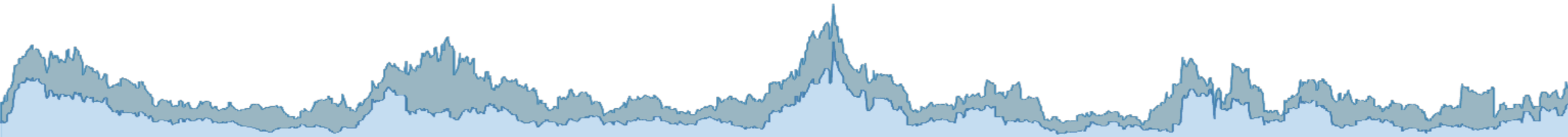
No Tradeoffs

Hyper makes no tradeoffs when it comes to **ACID guarantees** and **SQL-92+ language support (based on PostgreSQL dialect)** that stood the test of time.

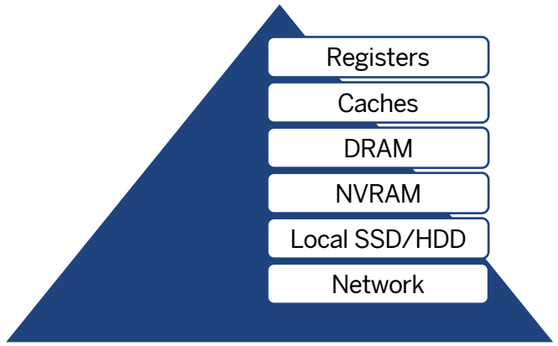


No Delays

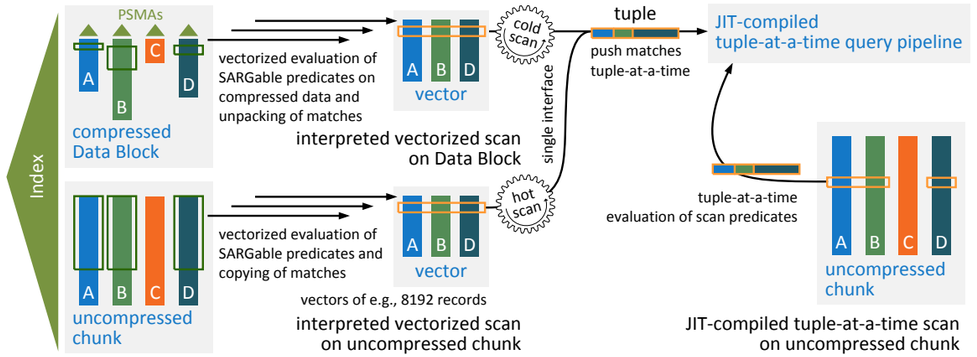
Hyper **scales with available hardware resources** to allow **highest performance on all workload classes.**



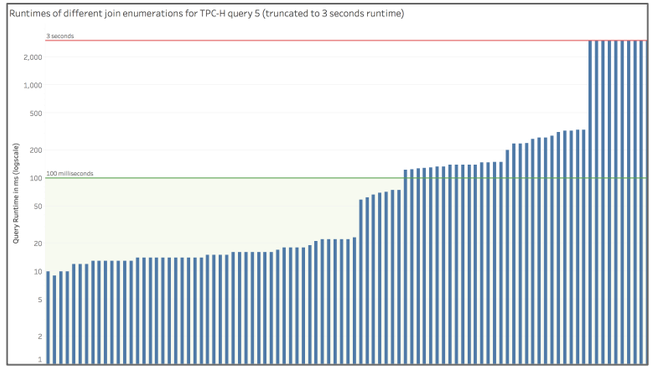
Inside Hyper



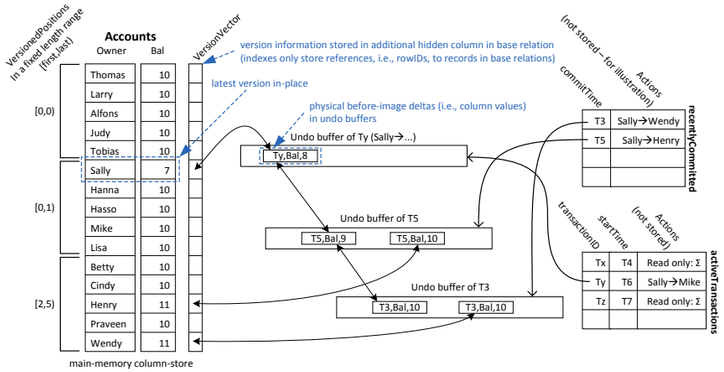
Optimized for Storage Hierarchy



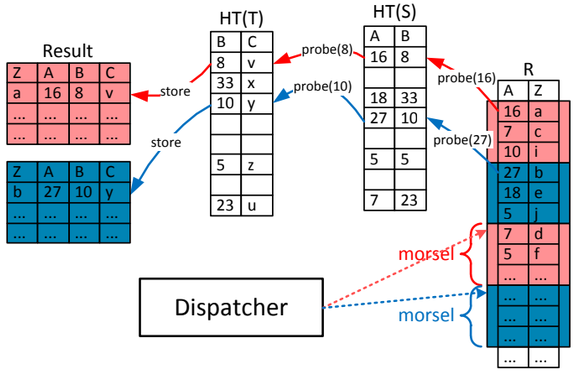
Query Compilation & Vectorized Scans



Advanced Query Optimization



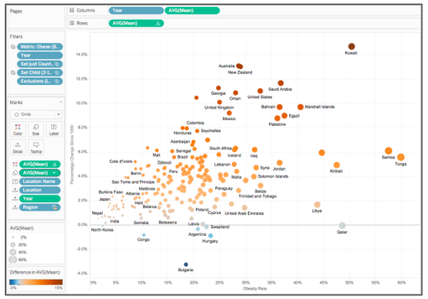
Fast MVCC

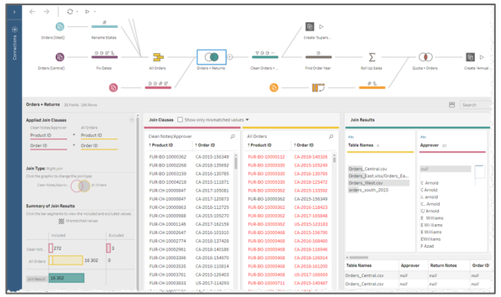


Morsel-Driven Parallelization

Hyper in Tableau

Desktop Online Public Server Prep



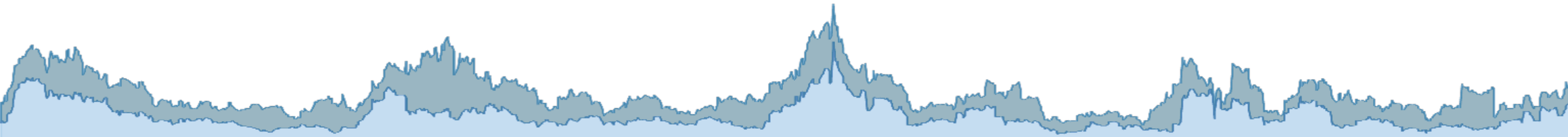


SQL Queries Results

Hyper

(Bulk) Insert, Update, Delete, Streams, Files

Extract Hyper API Federation Prep



From HyPer to Hyper: Challenges

Support

- Limited support provided up to 30 months after major product version release
- Compare performance across releases and database engines
- Semantic differences

Infrastructure

- Windows, Linux, and macOS
- Small laptops to large-scale servers and Cloud deployments

Workload

- Long tail of query complexity generated by Tableau
- Wide variety of data set characteristics

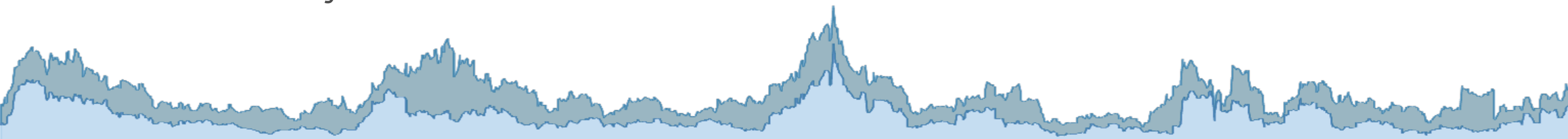
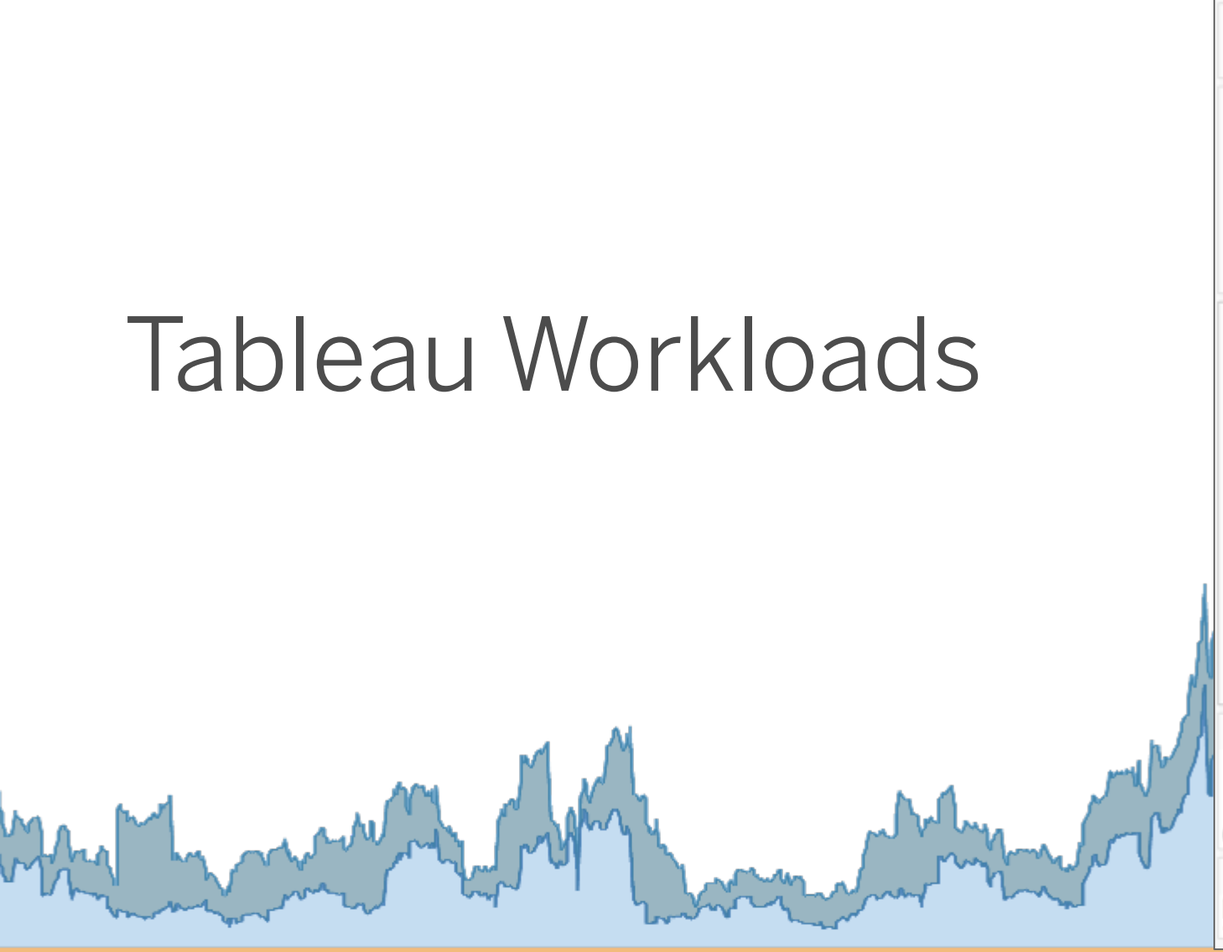
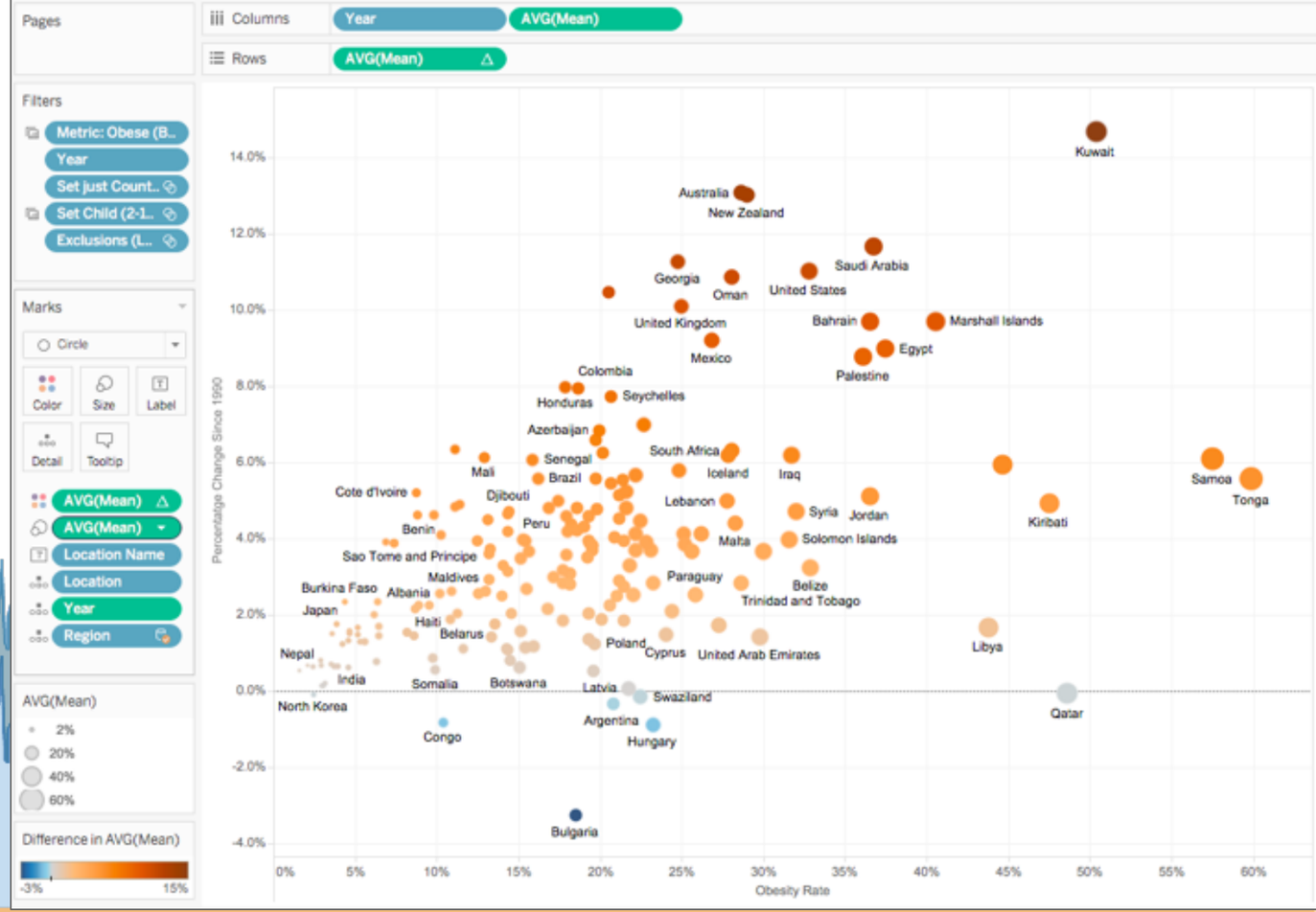
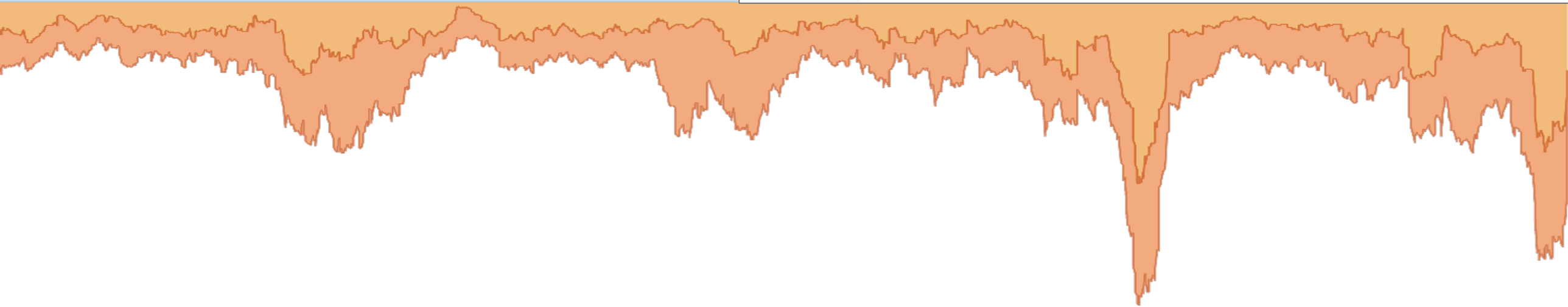


Tableau Workloads



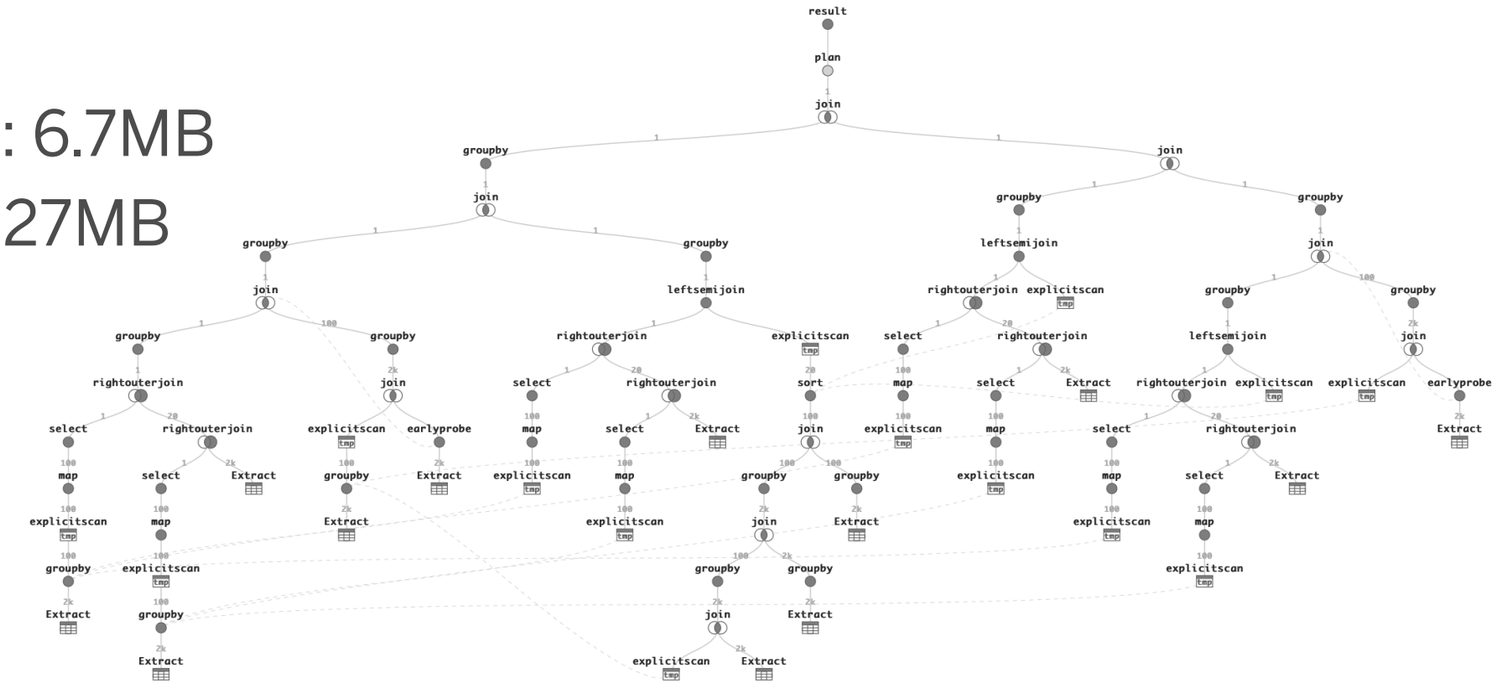


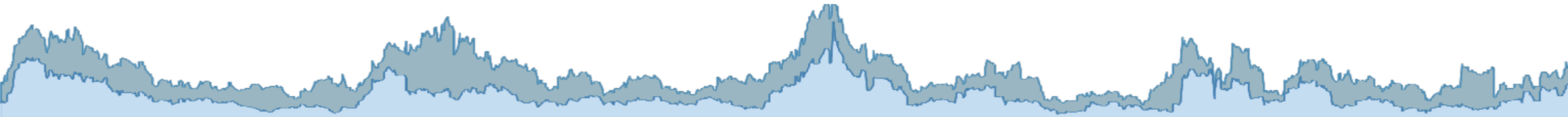


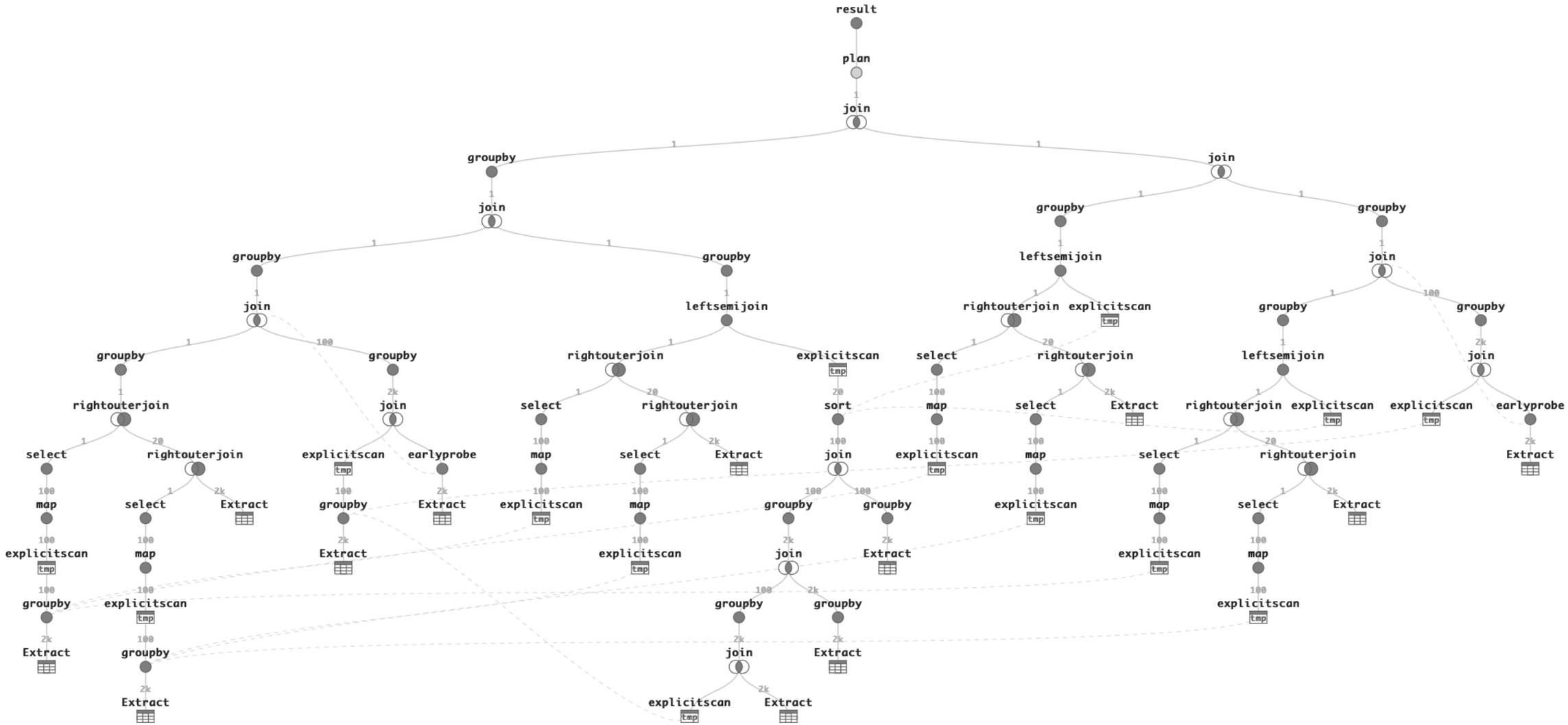
What Tableau Workloads Look Like

- Most queries are “small”: Only 0.5% larger than 5KB SQL Text
- But: **Huge** outliers
- Largest query in our data set: 6.7MB
- Largest query we saw so far: 27MB

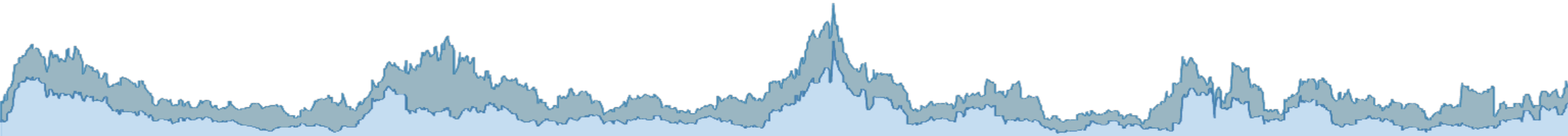
And that’s not all due to constant strings…



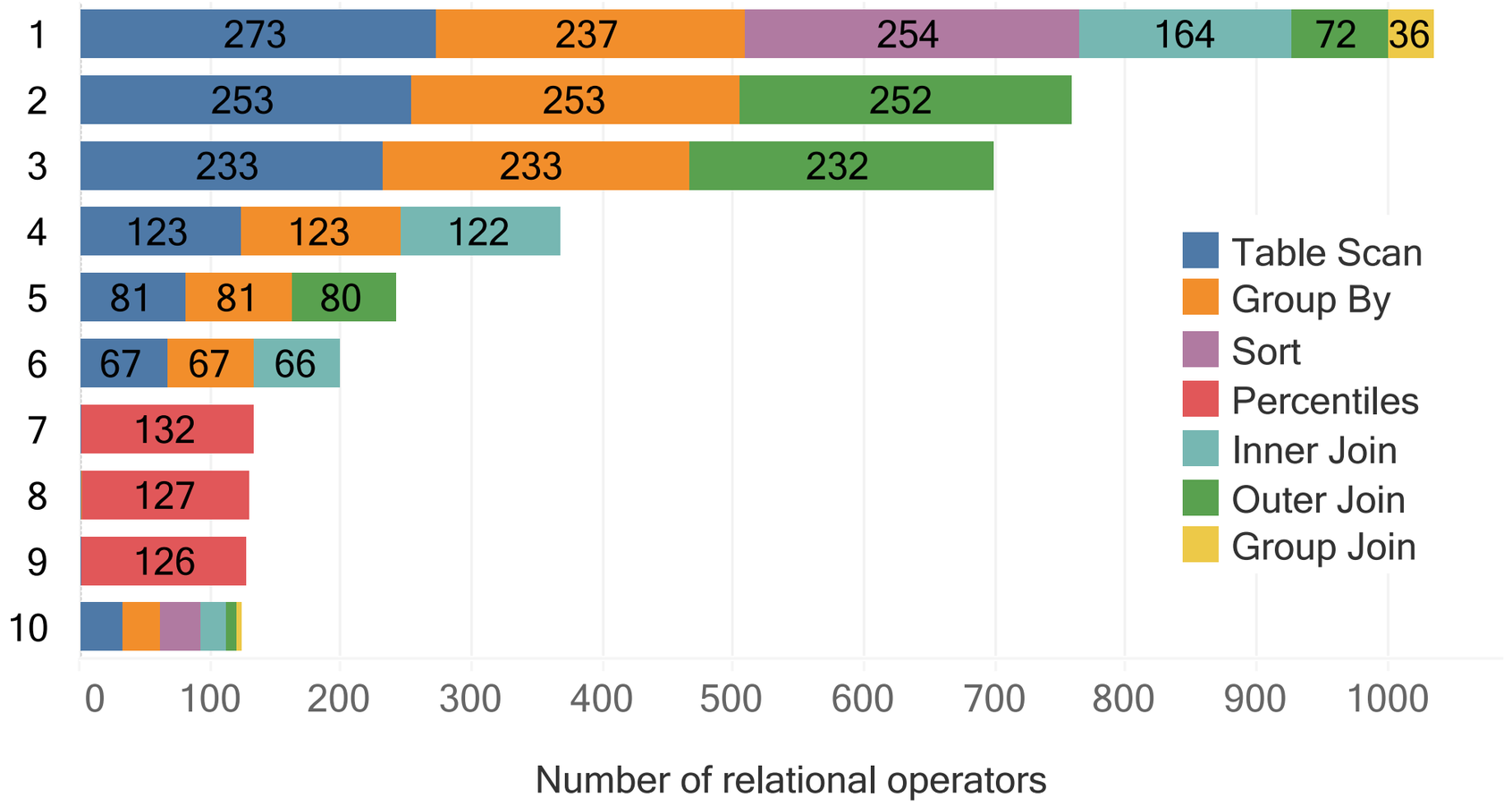




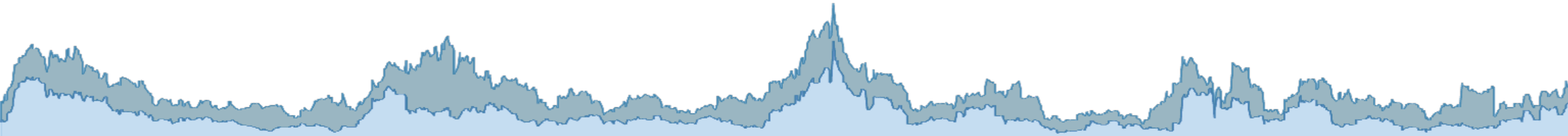
Need a query plan visualizer? https://github.com/tableau/query-graphs/

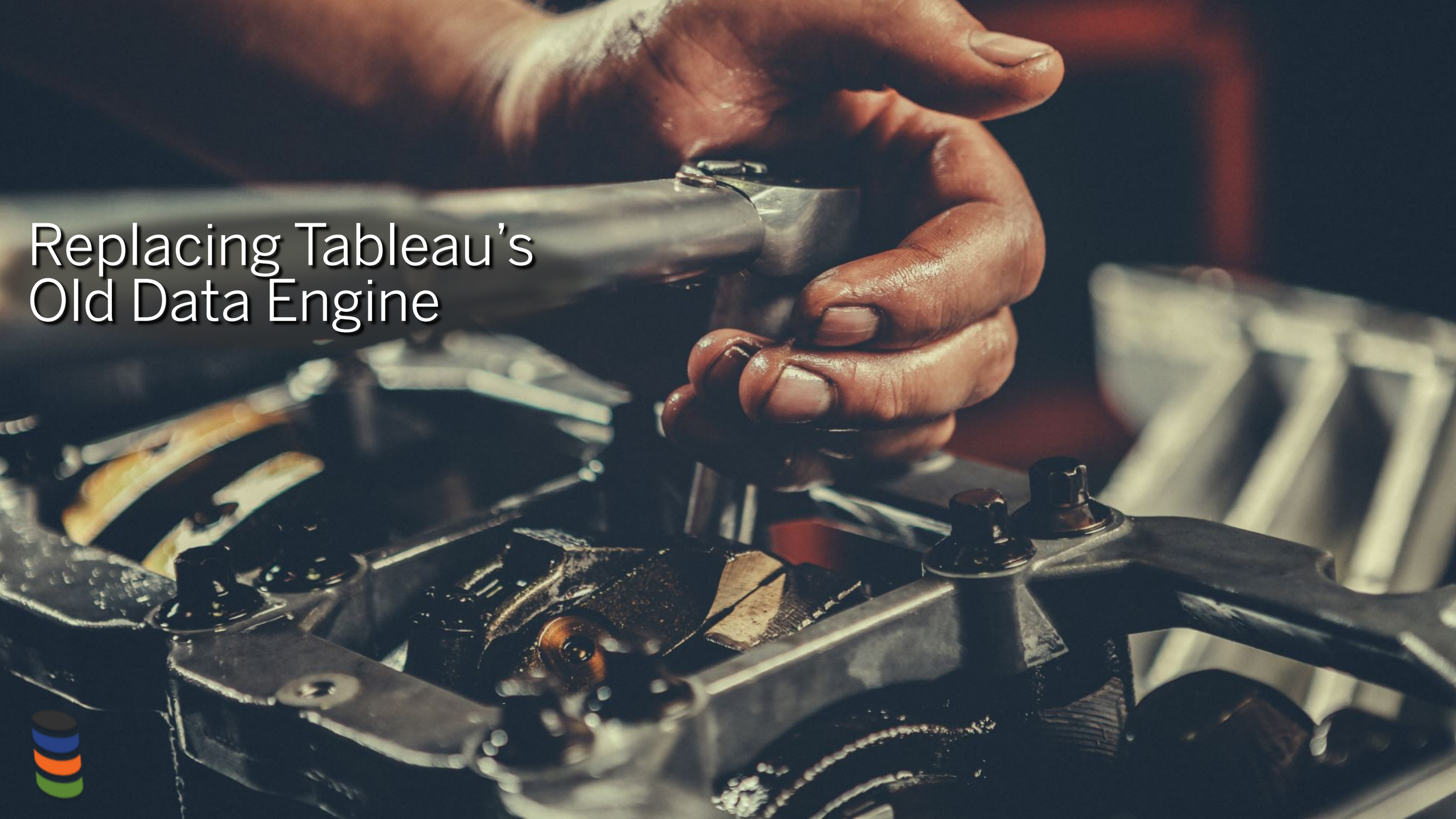


What Tableau Workloads Look Like



Vogelsgesang et al.: Get Real: How Benchmarks Fail to Represent the Real World. *DBTest'18*




Replacing Tableau’s
Old Data Engine

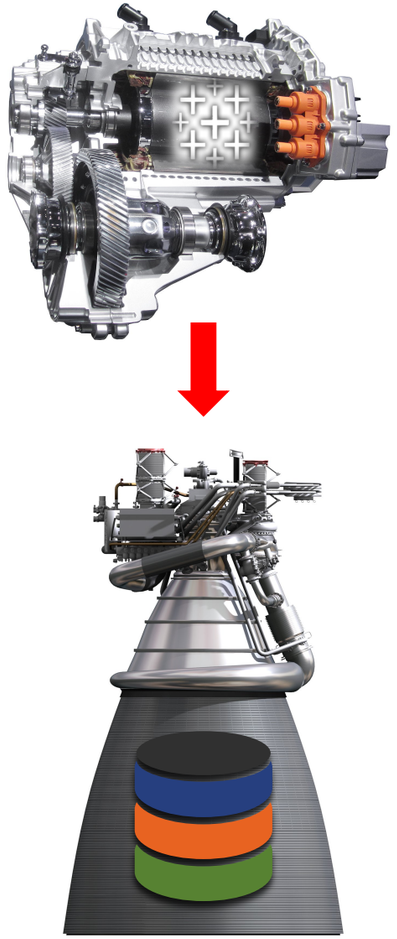
Replacing Tableau's Old Data Engine

Tableau's old data engine (TDE):
vector-based engine inspired by MonetDB/X100

First step: Replace TDE as the backend of all Tableau Products

Goals:

- Deliver performance at scale
- Seamless transition for customers

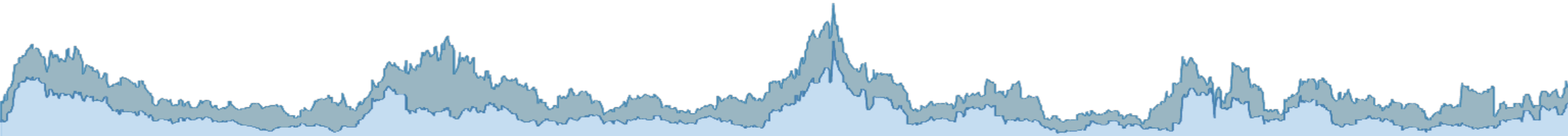


TDE: The Gold Standard

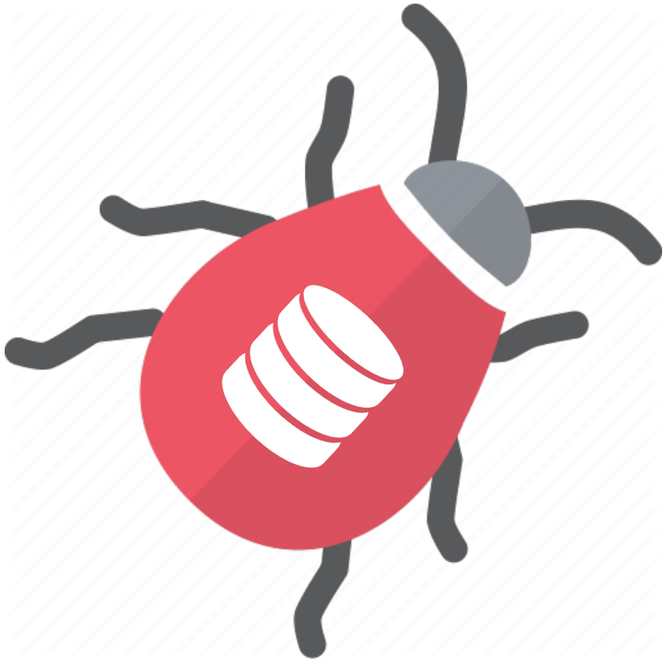
Having a gold standard is great!

We just ran a lot (60k) of workbooks from Tableau Public and compared results.

Simple, measurable goal:
Get results to match for all and be fast, then we're done ☺



Challenge: Bug Compatibility?

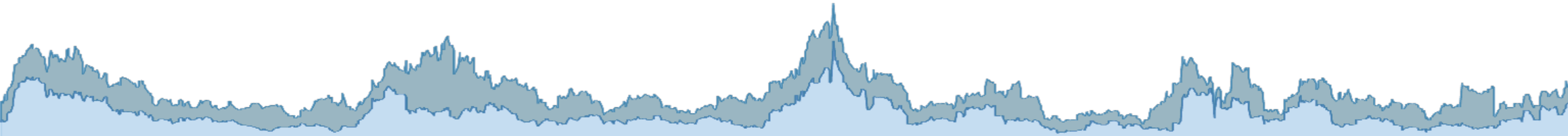


Is it really worth to show the same result for all queries?

- Non-deterministic behavior (parallelization!)
- Bugs in TDE

Our attitude changed over time:

1. 100% same results at all cost, customers don't want their Viz to change! This is non-negotiable!
2. Well, but what if it changes to be correct?
3. Who said that Visualizations can't change in the first place?
4. Let's do the right thing and fix things once and for all!



Compatibility Curiosity: String to Date Cast

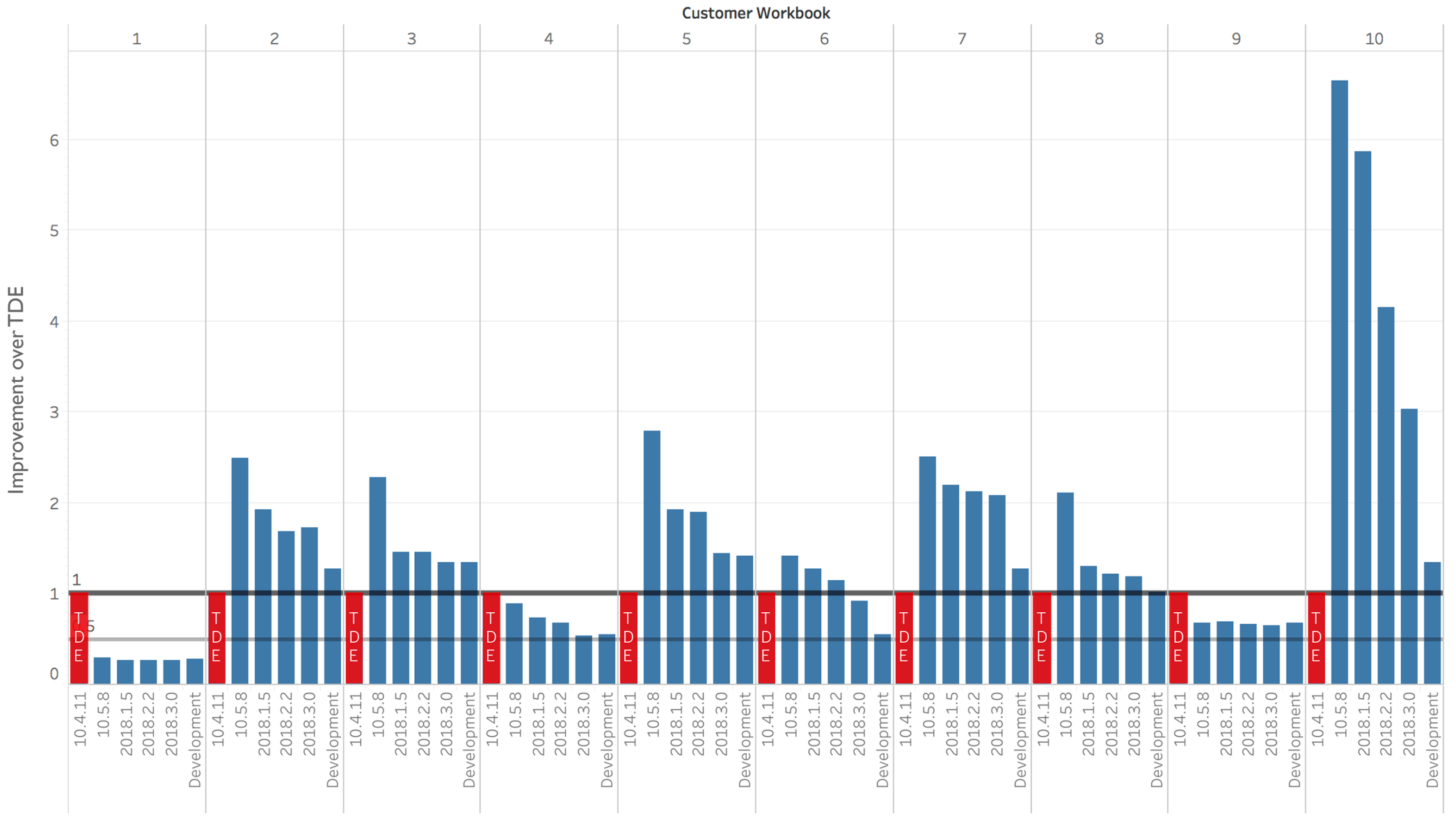
Is "**5/7/2020**" April 7 or July 5?

TDE's strategy: be aggressive finding a valid date. Sounds great!

But horrible in the relational model!

Input	TDE	Hyper
5/7/2020	April 7, 2020	April 7, 2020
15/7/2020	July 15, 2020	NULL

Silent failure: Sales workbook: More sales in the first 12 days of each months!



Continuous performance improvements past launch

Testing Hyper

SQL Level Testing

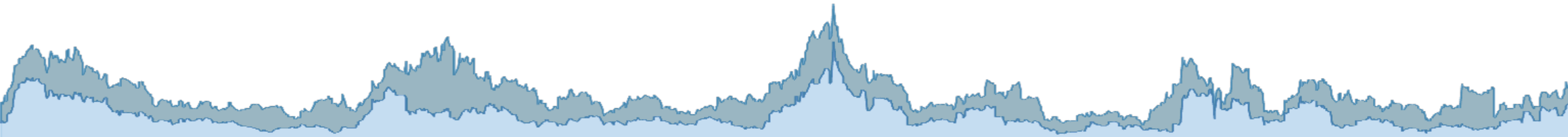
We started with the SQLite test suite

- Added own test cases for features
- Added regression tests for defects
- Added fine grained expectations (e.g., constant folding)

```
query N expectConstantResult
SELECT DATE '2001-09-28' + INTEGER '7'
----
2001-10-05

# DATE + INTEGER -> DATE with overflow
statement error 22003
SELECT DATE '4713-01-01 BC' - 1
```

How to execute the tests?

- First: Own Hyper front end that parses the file and executes the queries:
  - Problem: Server / protocol code not tested
- Second: Client (based on libpq) that parses the file and sends queries to a server
  - Problem: Harder to debug, test driver is not same process



Testing even more with SQL Level Tests

Use EXPLAIN statement to test optimizer

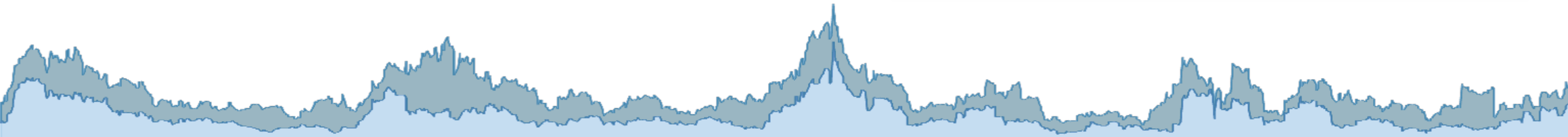
Introduce function to scan the own log to test for log messages

- Introduce trace settings that allow printing specific internals to the log

Special test functions with side effects to test further internals

- E.g., a function that allocates thread-local memory
- **SELECT suicide()**

```
# Deduplication for simple domain queries
query S
EXPLAIN SELECT * FROM
(SELECT a FROM t GROUP BY 1) t1,
(SELECT a FROM t GROUP BY 1) t2
----
          executiontarget(1)

                join(2)
                bnl

explicitscan(3) explicitscan(6)->(4)

  groupby(4)

 tablescan(5)
 t
```



SQL vs. C++ Unit Tests

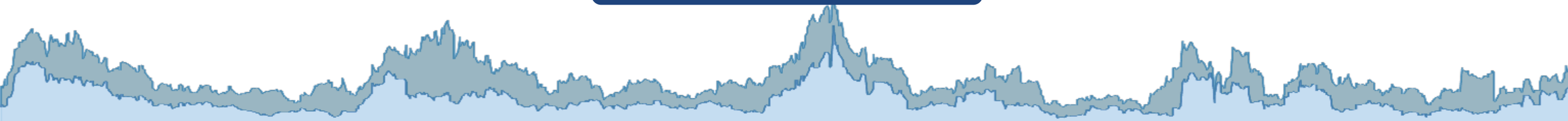
SQL

- Easy to write, usually very succinct
- No recompilation needed
- In vivo: Can run code in specific query contexts
- Good test failure reporting
- Can update expected test results automatically
- Resembles customer usage of the system

C++

- Only test the code in question, not the whole SQL layer
- Runs faster

Big controversy!



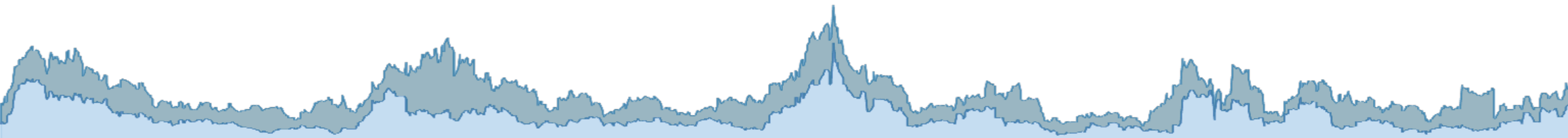
Beyond SQL Testing

SQLite tests are great, but they **can't simulate load** from multiple connections

Solution: Loadtest DSL

- open connection
- Embed SQLite test statement
- Execute code blocks in parallel
- Loops

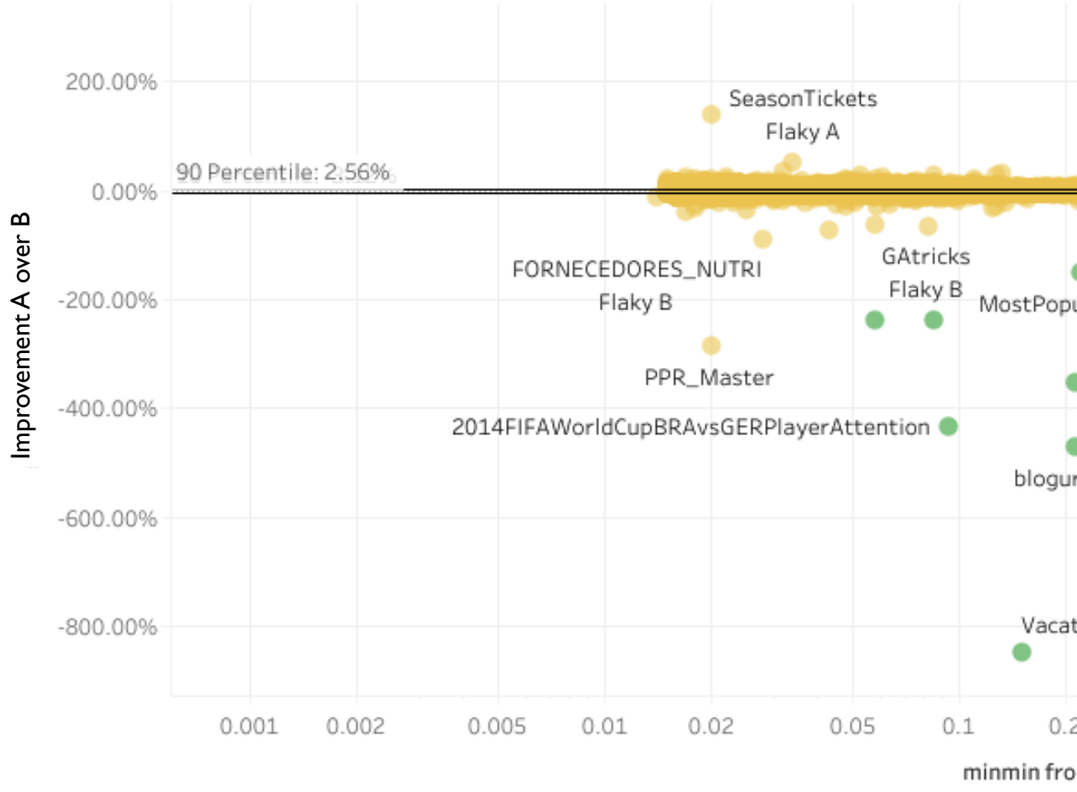
```
exec CREATE DATABASE mytestdb;

connection mytestdb user=bob {
  repeat 100 {
    exec CREATE TABLE foo AS
         (SELECT x FROM
          generate_series(1,1e6) x);
    parallel 10 {
      exec UPDATE foo SET x = x+10
           WHERE x % 2 = 0;
    } and 2 {
      test {
        query N
        SELECT SUM(x) FROM foo
        WHERE x % 2 = 1;
        ----
        1234567
      }
    }
  }
```

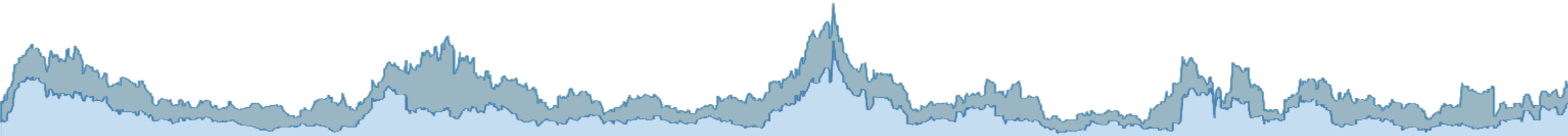


A/B Testing: MaxPerf / QueryRunner

1. Goal: Check for compatibility with TDE
   I. Correctness and perf
   II. A/B Test on the 60k Public Workbooks

2. How it works
   I. Starts up Hyper
   II. Loads a workbook
   (thus sending queries to Hyper)
   III. Checks number of marks
   / mark values and records times
   IV. A/B test between old and new branch



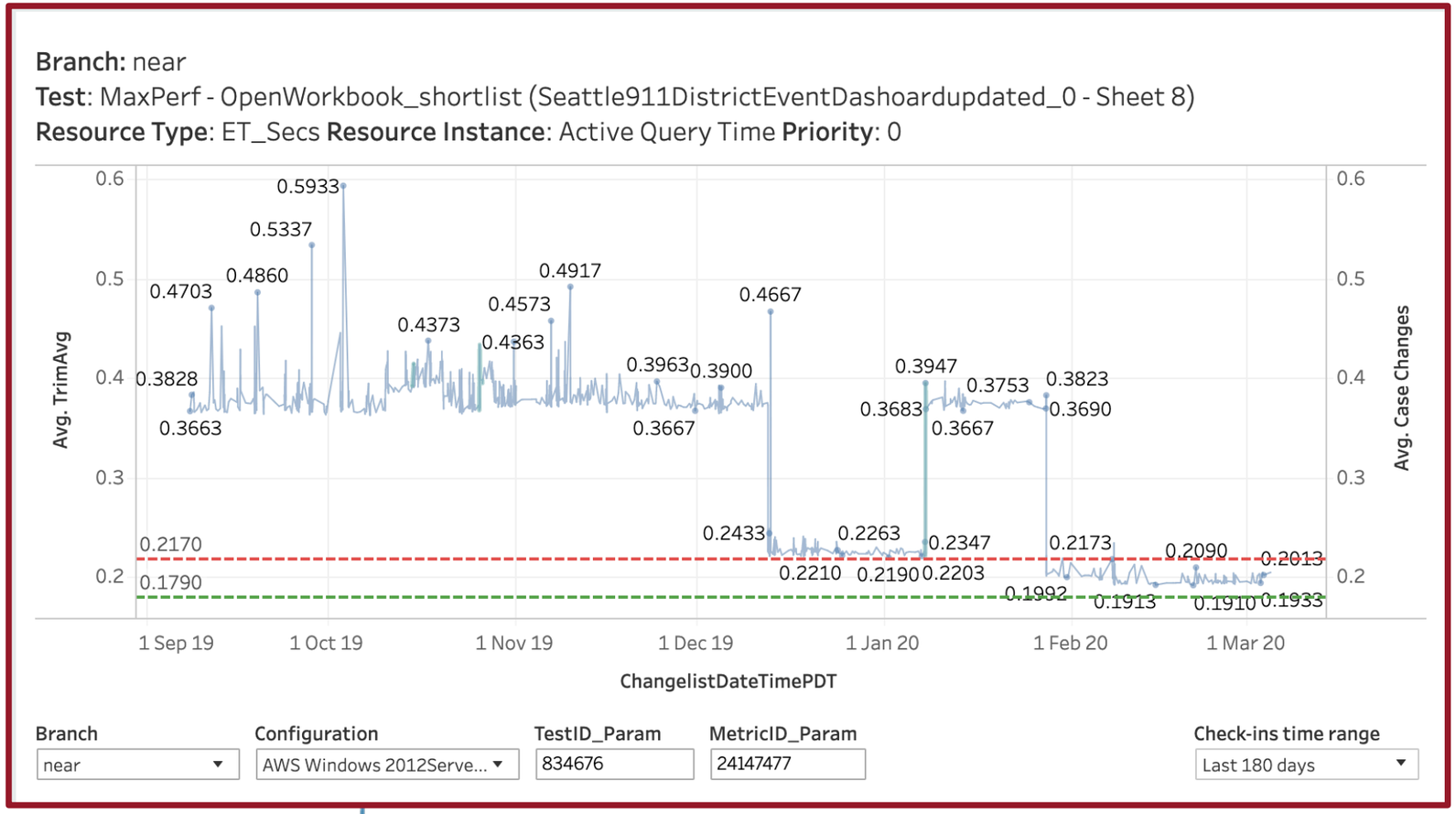


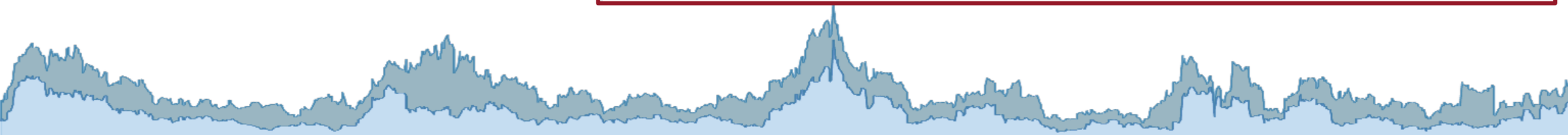
Automatic Regression Testing

1. Measure perf on every commit
2. If perf regresses, file a defect
3. Make sure it's not just noise
4. If perf improves, make it the new expectation

DEFECT 1023800

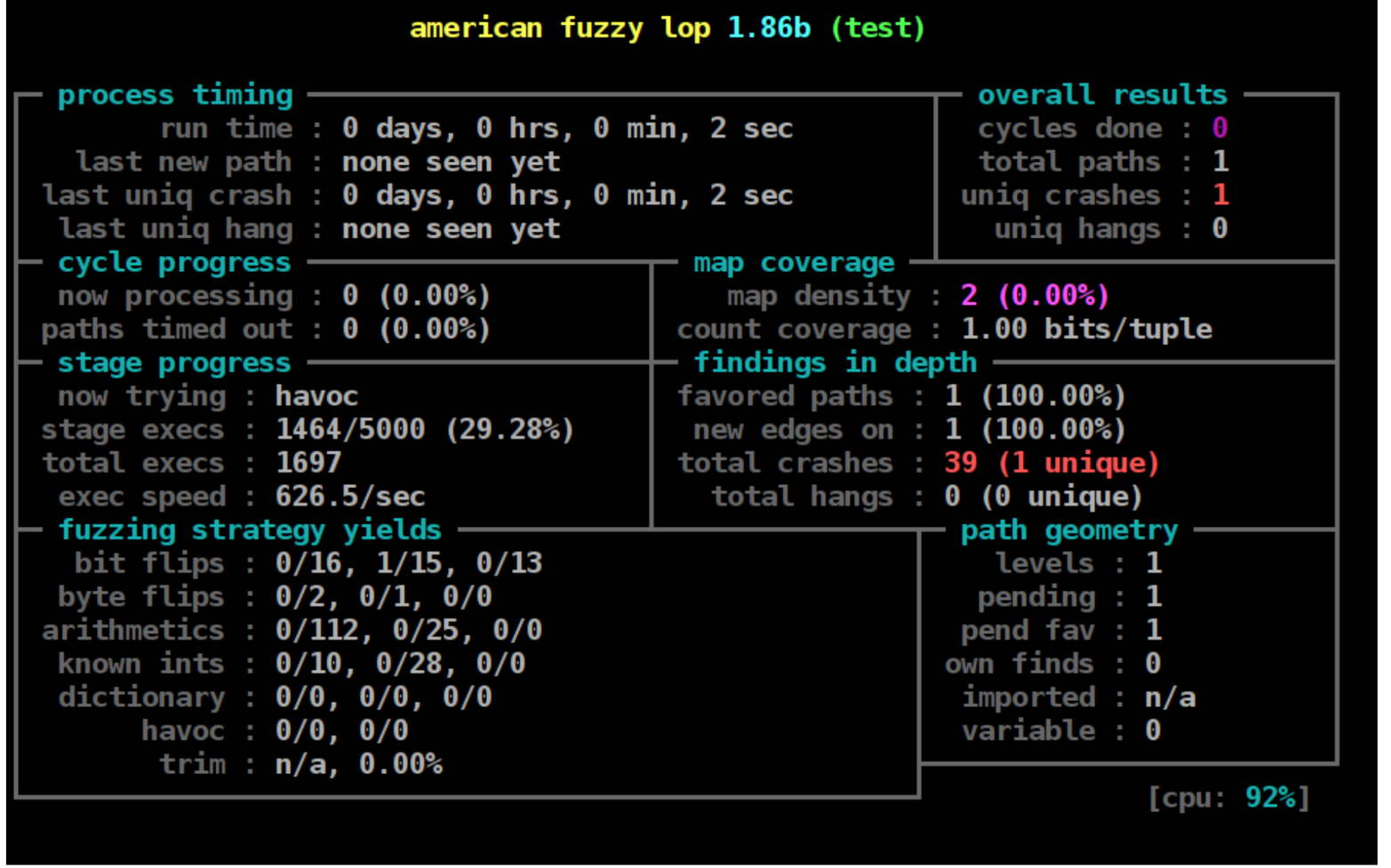
1023800 Perf: 21 cases regressed by up to 356.67ms (up to 32%) starting from CL 1982374, near.19.1025.1835 🦆≠☠️



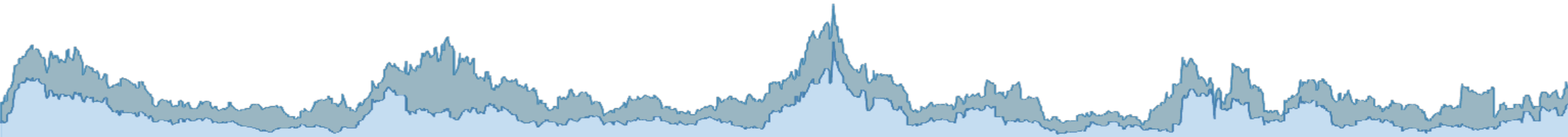


Fuzzing

1. Use a fuzzer (e.g., AFL)
2. Feed it the SQL grammar
3. Let it run for a long time



Found several vulnerabilities and defects with fuzzing!



Static Code Analysis

1. Enable all compiler warnings and make them errors
   –Wall –Wextra –Werror -Woverloaded-virtual -Wunreachable-code-return…

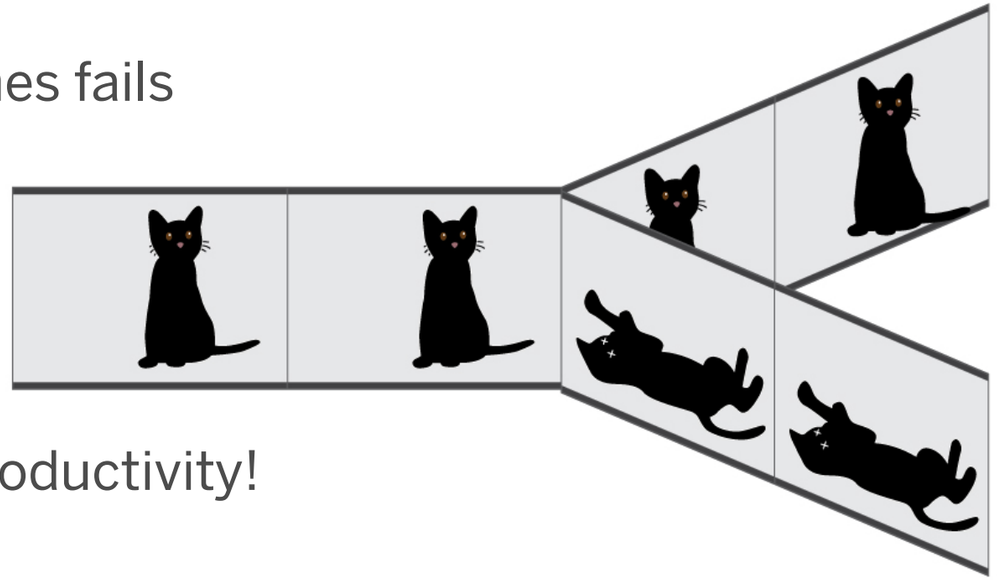
2. Keep code clean: clang-format, clang-tidy

3. Clang static analyzers
   - Address sanitizer
   - Thread sanitizer and Memory sanitizer
     - Third party libraries must also be re-built!
   - Undefined behavior sanitizer
   - …

4. Code coverage

Flaky Tests: The root of all (test) evil

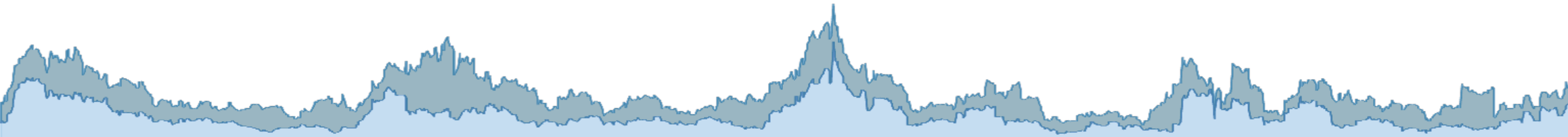
Flaky test: A test that sometimes passes and sometimes fails
Worst cases: It succeeds 99,9% of the time



- Don’t let flaky tests build up!
- Don’t get into the habit of muting flaky tests!
- Treat them as a high priority defects! They kill dev productivity!

Root causes:

- Real defect
- Bad test; usually dependent on timing or other



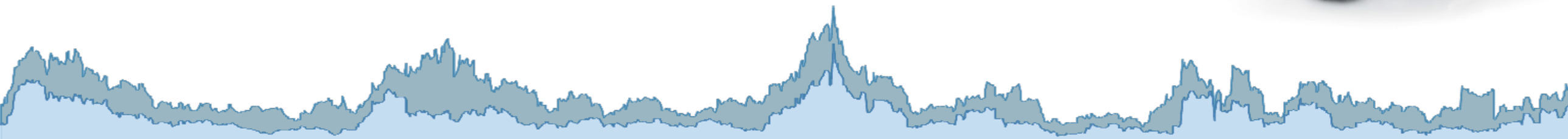
Lessons Learned

Benchmark Responsibly!

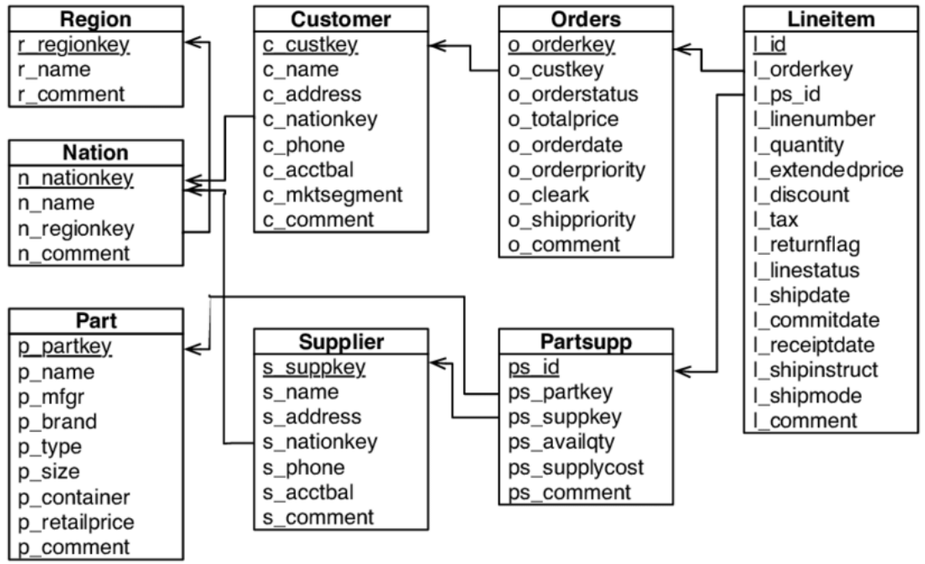
When benchmarking a system …

- Contact the vendor
- Report bugs
- Share your benchmark ahead of time, if possible
  - Allows vendors to give feedback, double check the validity
  - Good chance to increase the quality of your benchmark
- Ask the vendor how to configure the system
  - Don’t let misconfigured benchmarks impact the credibility of your hard work

- Shout-out to Andrew C. from Brown University



There is more than TPC-X

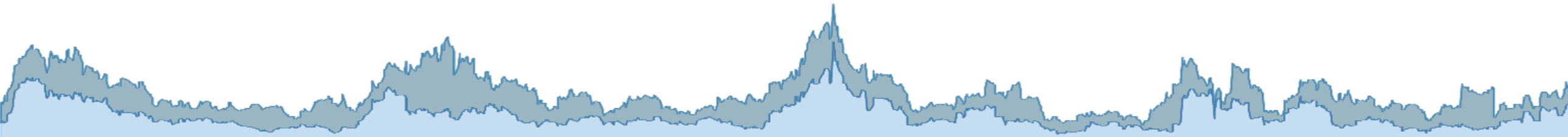


Generated workloads are real

I. More and more tools generate queries
II. Queries are way more complex than hand-written TPC-X queries
III. Plenty of interesting (and novel!) problems lurking in other workloads

Public BI Benchmark from CWI

- https://github.com/cwida/public_bi_benchmark
- Ghita, Tomé, Boncz, **White-box Compression: Learning and Exploiting Compact Table Representations**, *CIDR'20*
- Based on Tableau Public data



Build your system for production

Build a system; not a throw-away prototype

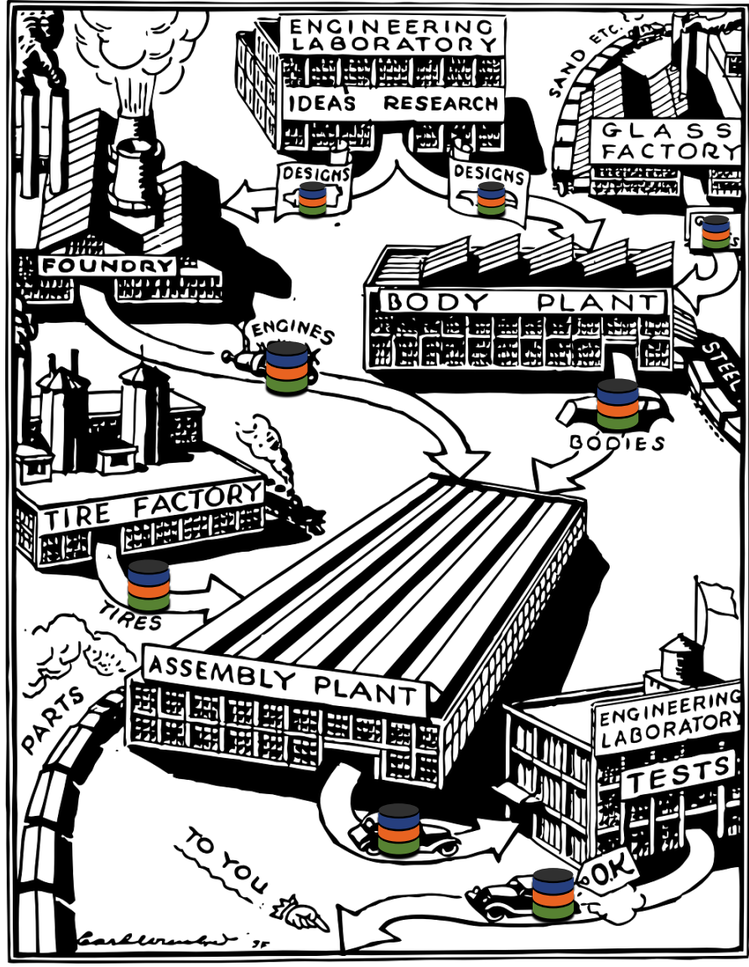
- Much more rewarding, longer lasting sense of achievement
- Easier to build upon previous work
- Results closer to reality (micro benchmarks leave out crucial parts)

Follow standards (e.g., SQL, PostgreSQL)

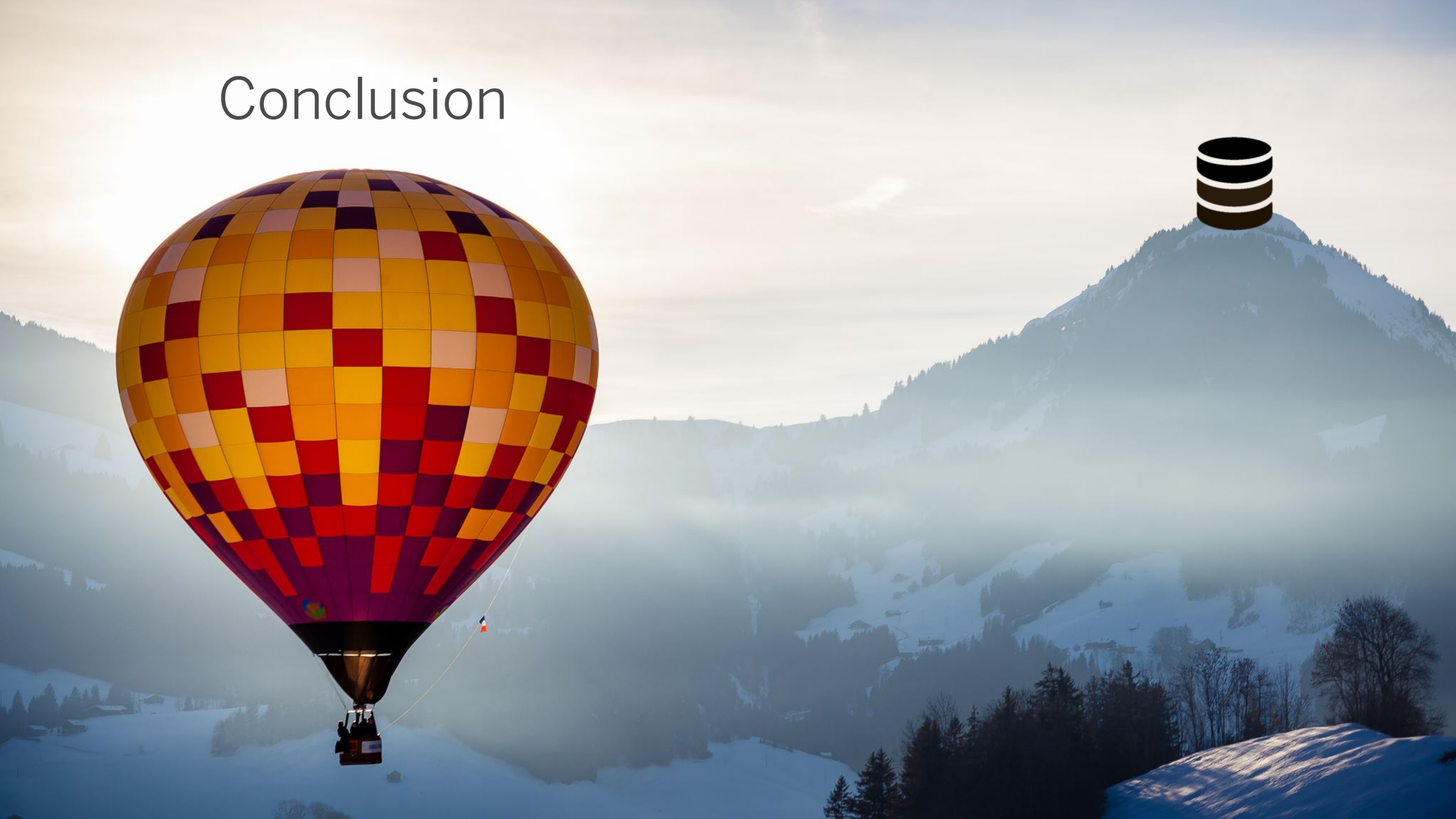
- Easy test adaption, easier benchmarking

Build that system **as if it was for production**

- Test driven development is great;
  defects could make your results invalid!
- Easier to adapt a stable system to the next benchmark
- There is a real chance to get your system into production in the end
- OS compatibility (Some perf hacks are highly non-portable!)



Conclusion



Conclusion

Academic Projects can make it into production

- Design to become a product (it's fun!)
- Replacing an old system: great gold standard, hard to beat in all cases

TPC-X is not everything!

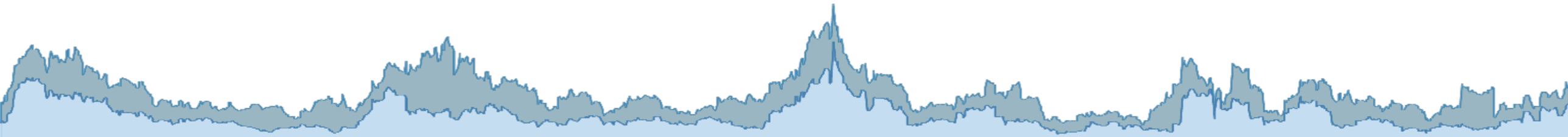
- Experiment with more diverse workloads, real world queries are complex

Various layers of testing required

- Don't forget stress testing, don't regress, and avoid flaky tests

Benchmark responsibly

Try out Hyper API! tabsoft.co/hyperapi




tableau®