Lecture #10

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Vectorization vs. Compilation

Andy Pavlo // 15-721 // Spring 2023

ADMINISTRIVIA

Project #1: Sunday February 26th

Project #2: Sunday April 30th

Project #3
→ Proposals: Wednesday March 1st
→ Updates: Monday April 3rd
→ Final Presentations: TBA

OBSERVATION

Vectorization can speed up query performance.

Compilation can speed up query performance.

We have not discussed which approach is better and under what conditions.

Switching an existing DBMS is difficult, so one must make this design decision early.

VECTORWISE – PRECOMPILED PRIMITIVES

Pre-compiles thousands of "primitives" that perform basic operations on typed data.

→ Using simple kernels for each primitive means that they are easier to vectorize.

The DBMS then executes a query plan that invokes these primitives at runtime.

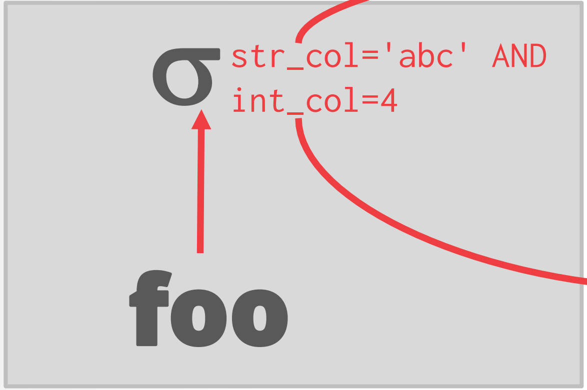
→ Function calls are amortized over multiple tuples.

→ The output of a primitive are the offsets of tuples that satisfy the predicate that the primitive represents.

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

VECTORWISE – PRECOMPILED PRIMITIVES

```sql
SELECT * FROM foo
 WHERE str_col = 'abc'
   AND int_col = 4;
```




```
vec<offset> sel_eq_str(vec<string> batch, string val) {
  vec<offset> res;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] == val) res.append(i);
  return (res);
}
```

```
vec<offset> sel_eq_int(vec<int> batch, int val,
                       vec<offset> positions) {
  vec<offset> res;
  for (offset i : positions)
    if (batch[i] == val) res.append(i);
  return (res);
}
```

HYPER – HOLISTIC QUERY COMPILATION

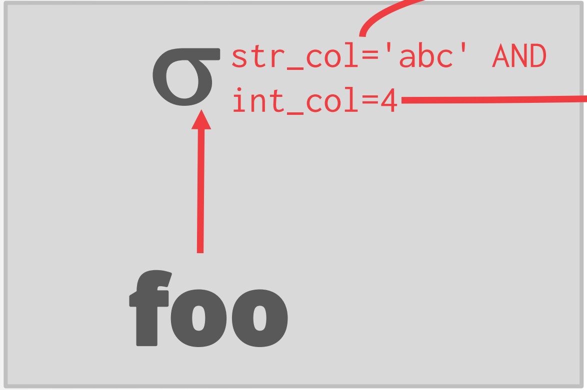
Compile queries in-memory into native code using the LLVM toolkit.

Organizes query processing in a way to keep a tuple in CPU registers for as long as possible.
→ Bottom-to-top / push-based query processing model.
→ Not vectorizable (as originally described).

EFFICIENTLY COMPILING EFFICIENT QUERY PLANS FOR MODERN HARDWARE
VLDB 2011

HYPER – HOLISTIC QUERY COMPILATION

```sql
SELECT * FROM foo
 WHERE str_col = 'abc'
   AND int_col = 4;
```




```
vec<offset> scan_row(vec<string> str_col, string val0,
                     vec<int> int_col, int val1) {
  vec<offset> res;
  for (offset i = 0; i < str_col.size(); i++)
    if (str_col[i] == val0 && int_col[i] == val1)
      res.append(i);
  return (res);
}
```

TODAY'S AGENDA

Vectorization vs. Compilation
Project #2
Project #3

VECTORIZATION VS. COMPILATION

Test-bed system to analyze the trade-offs between vectorized execution and query compilation.

Implemented high-level algorithms the same in each system but varied the implementation details based on system architecture.

→ Example: Hash join algorithm is the same, but the systems use different hash functions (Murmur2 vs. CRC32×2)

EVERYTHING YOU ALWAYS WANTED TO KNOW ABOUT COMPILED AND VECTORIZED QUERIES BUT WERE AFRAID TO ASK
VLDB 2018

IMPLEMENTATIONS

Approach #1: Tectorwise

→ Break operations into pre-compiled primitives.
→ Must materialize the output of primitives at each step.

Approach #2: Typer

→ Push-based processing model with JIT compilation.
→ Process a single tuple up entire pipeline without materializing the intermediate results.

TPC-H WORKLOAD

Q1: Fixed-point arithmetic, 4-group aggregation

Q6: Selective filters. Computationally inexpensive.

Q3: Join (build: 147k tuples / probe: 3.2m tuples)

Q9: Join (build: 320k tuples / probe: 1.5M tuples)

Q18: High-cardinality aggregation (1.5m groups)

TPC-H ANALYZED: HIDDEN MESSAGES AND LESSONS LEARNED FROM AN INFLUENTIAL BENCHMARK
TPCTC 2013

TPC

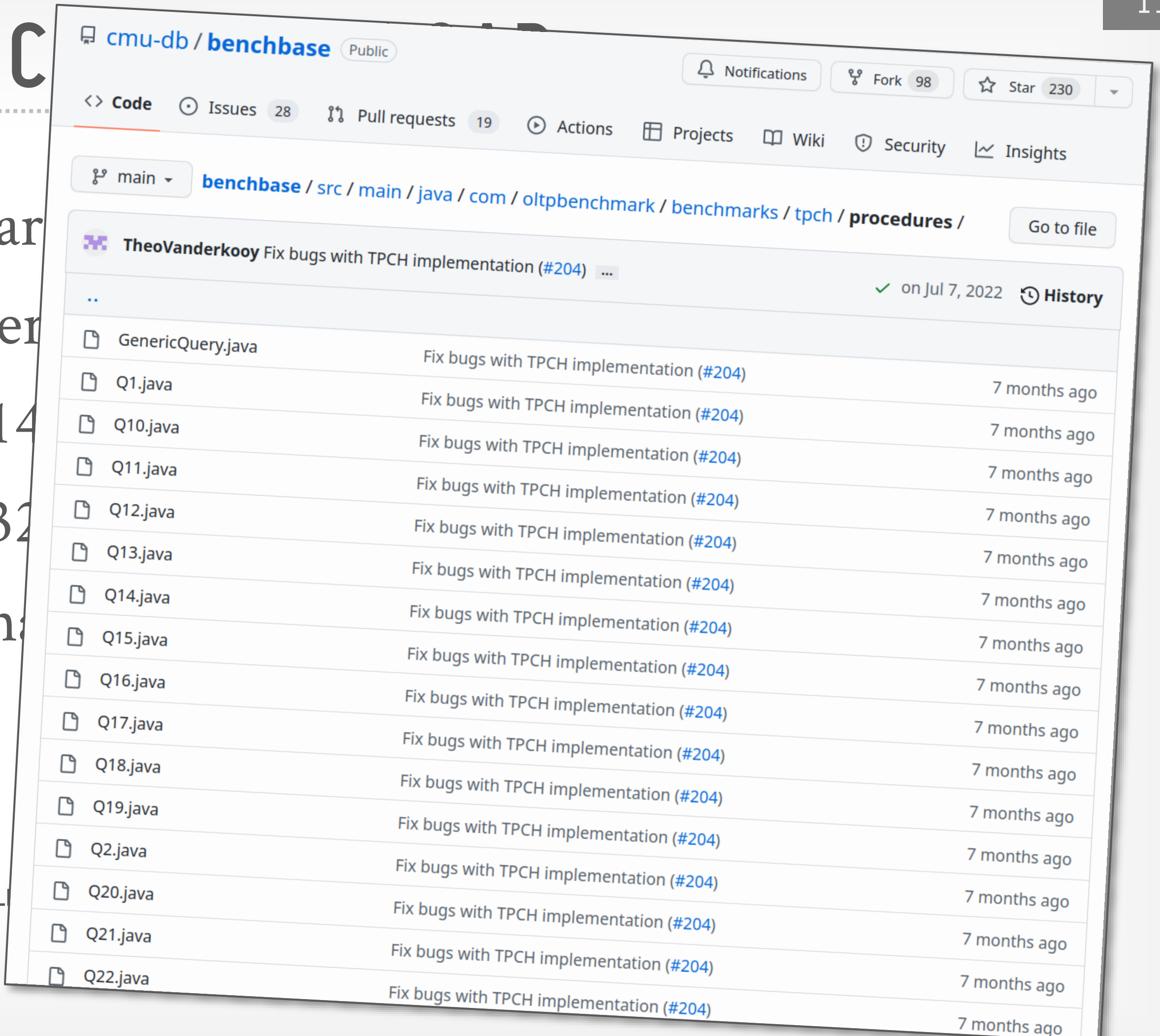
Q1: Fixed-point ar

Q6: Selective filter

Q3: Join (build: 14

Q9: Join (build: 32

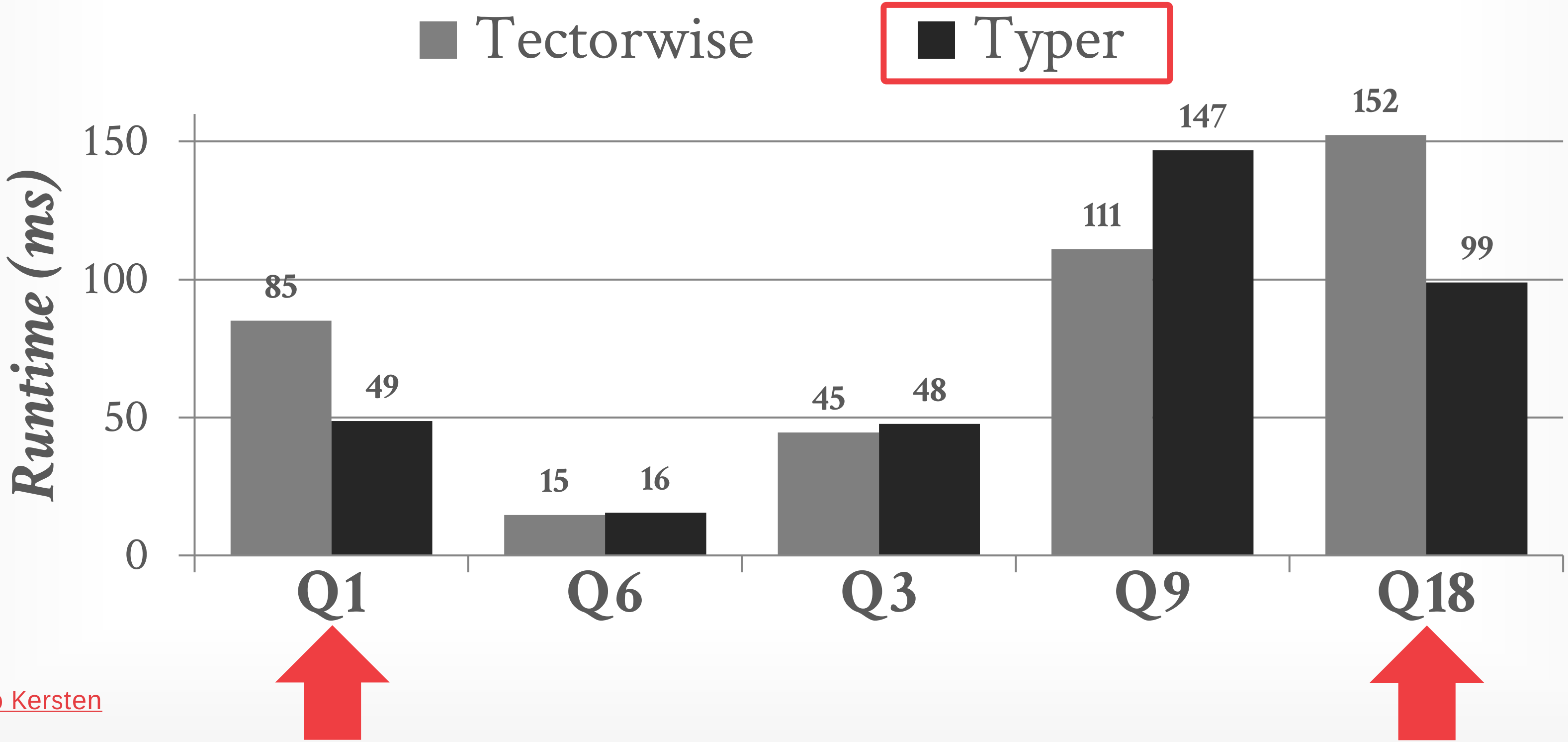
Q18: High-cardin



TPC-H ANALYZED: HIDDEN MESSAGES AND LESSONS L
FROM AN INFLUENTIAL BENCHMARK
TPCTC 2013

SINGLE-THREADED PERFORMANCE

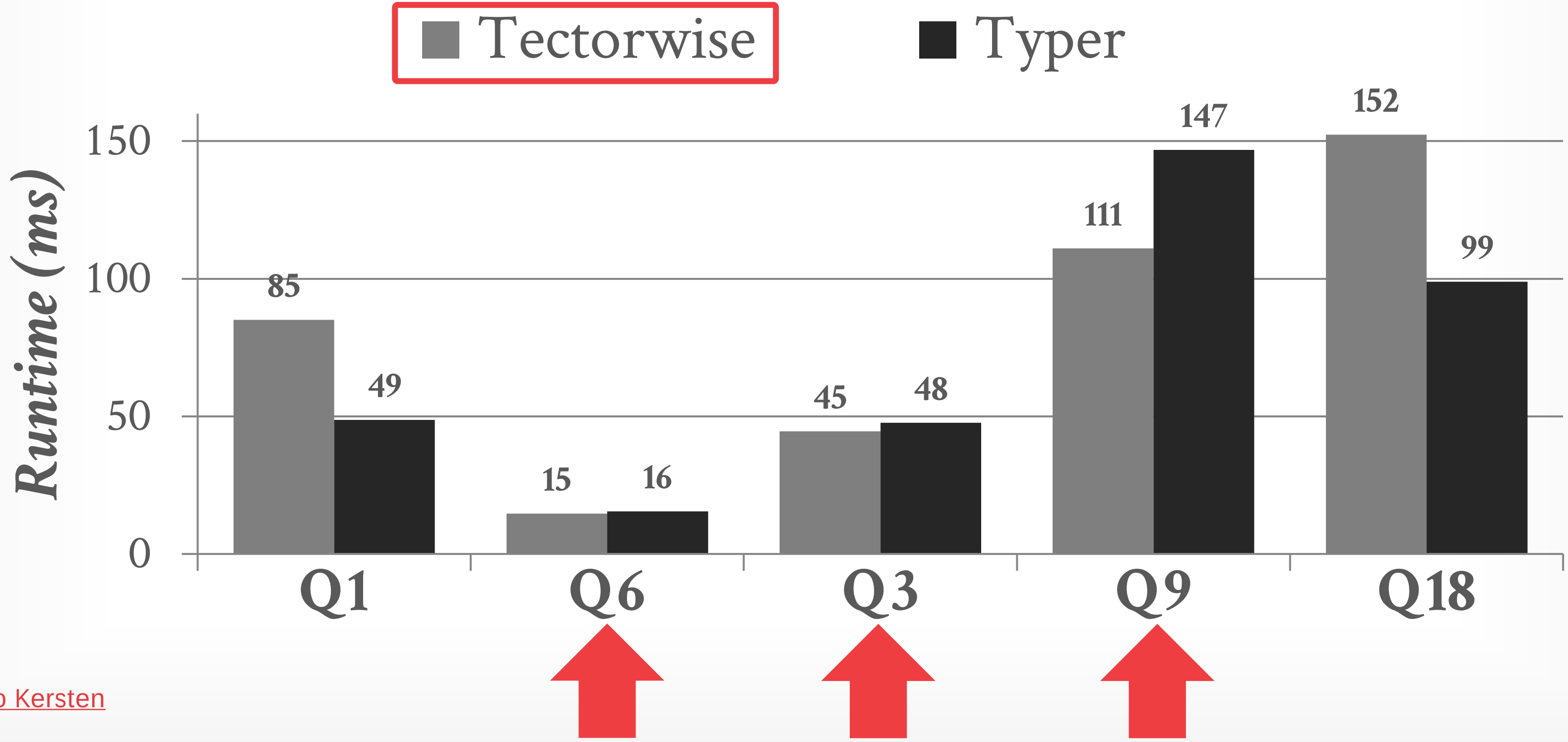
Intel Core i9-7900X (10 cores × 2HT)
TPC-H Queries (Scalefactor=1)



Source: Timo Kersten

SINGLE-THREADED PERFORMANCE

Intel Core i9-7900X (10 cores × 2HT)
TPC-H Queries (Scalefactor=1)



Source: Timo Kersten

SINGLE-THREADED PERFORMANCE

		Runtime	Cycles	IPC	Instr.	L1 Miss	LLC Miss	Br. Miss
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

SINGLE-THREADED PERFORMANCE

		Runtime	Cycles	IPC	Instr.	L1 Miss	LLC Miss	Br. Miss
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

SINGLE-THREADED PERFORMANCE

		Runtime	Cycles	IPC	Instr.	L1 Miss	LLC Miss	Br. Miss
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	★68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	★46	0.8	0.19	0.16

SINGLE-THREADED PERFORMANCE

		Runtime	Cycles	IPC	Instr.	L1 Miss	LLC Miss	Br. Miss
Q1	TWise	85	59	★2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	★2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

SINGLE-THREADED PERFORMANCE

		Runtime	*Cycles*	*IPC*	*Instr.*	*L1 Miss*	*LLC Miss*	*Br. Miss*
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	★1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	★21	0.5	0.16	0.27
Q9	TWise	111	56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

SINGLE-THREADED PERFORMANCE

		Runtime	*Cycles*	*IPC*	*Instr.*	*L1 Miss*	*LLC Miss*	*Br. Miss*
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	★ 0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

SINGLE-THREADED PERFORMANCE

		Runtime	*Cycles*	*IPC*	*Instr.*	*L1 Miss*	*LLC Miss*	*Br. Miss*
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	56	★1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	★42	1.7	0.46	★0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

SINGLE-THREADED PERFORMANCE

		Runtime	*Cycles*	*IPC*	*Instr.*	*L1 Miss*	*LLC Miss*	*Br. Miss*
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	★56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

MAIN FINDINGS

Both models are efficient and achieve roughly the same performance.
→ 100x faster than row-oriented DBMSs!

Data-centric is better for "calculation-heavy" queries with few cache misses.

Vectorization is slightly better at hiding cache miss latencies.

SIMD PERFORMANCE

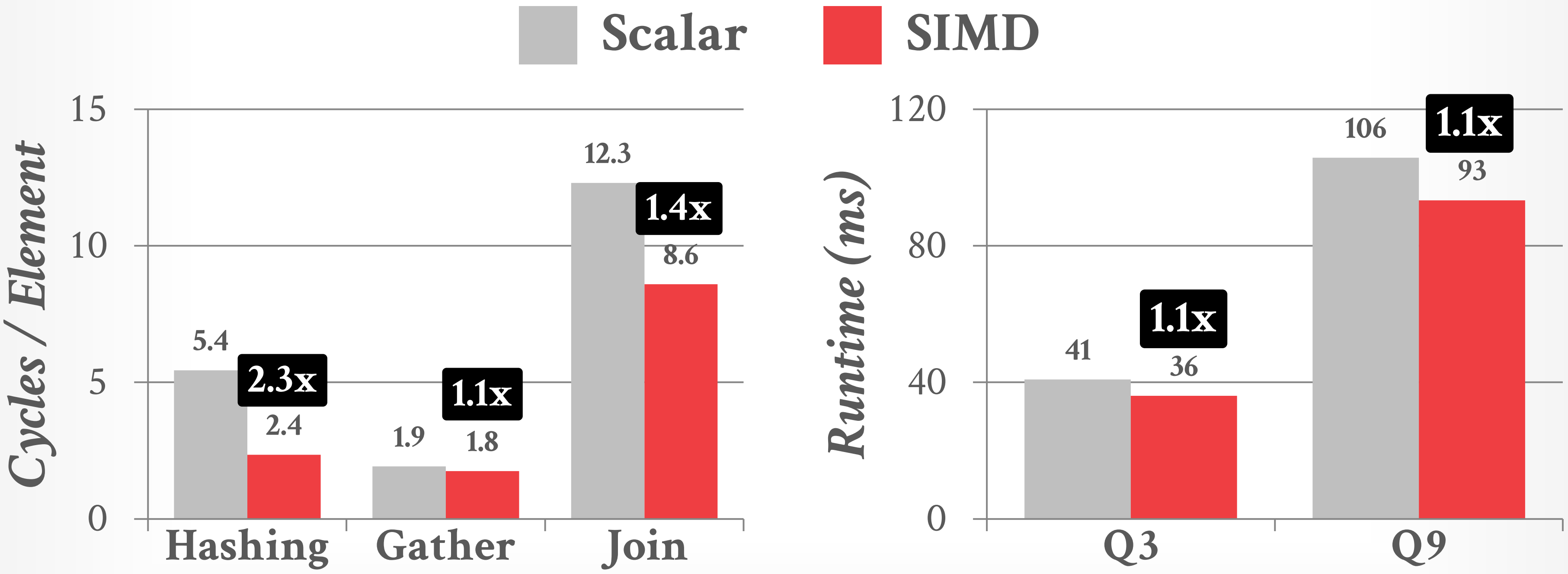
Evaluate vectorized branchless selection and hash probe in Tectorwise.

Use AVX-512 because it includes instructions to make it easier to implement algorithms using vertical vectorization.

→ Selective operations using bitmask registers.

SIMD EVALUATION`

Intel Core i9-7900X (10 cores × 2HT)
TPC-H Queries (Scalefactor=1)



Source: Timo Kersten

AUTO-VECTORIZATION

Evaluate how well the compiler can automatically vectorize the Vectorwise primitives.

→ Targets: GCC v7.2, Clang v5.0, ICC v18

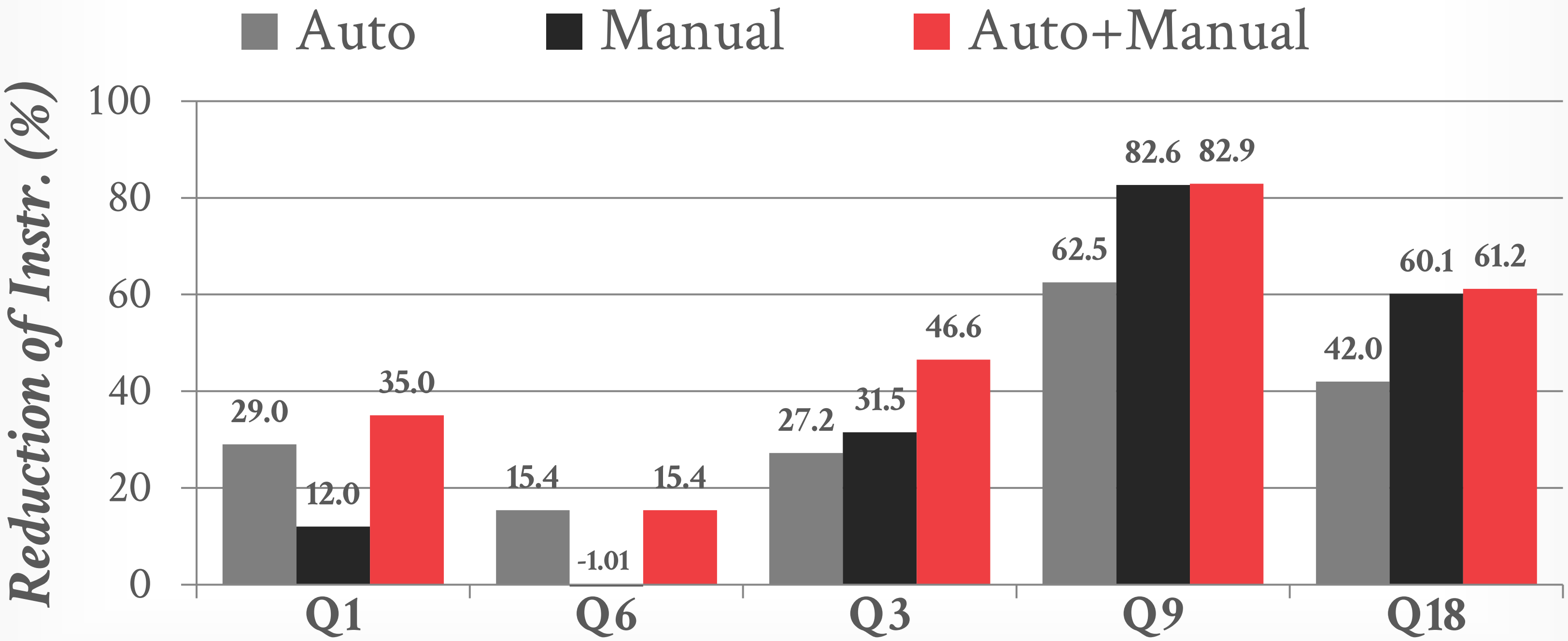
ICC was able to vectorize the most primitives using AVX-512:

→ Vectorized: Hashing, Selection, Projection

→ Not Vectorized: Hash Table Probing, Aggregation

AUTO-VECTORIZATION

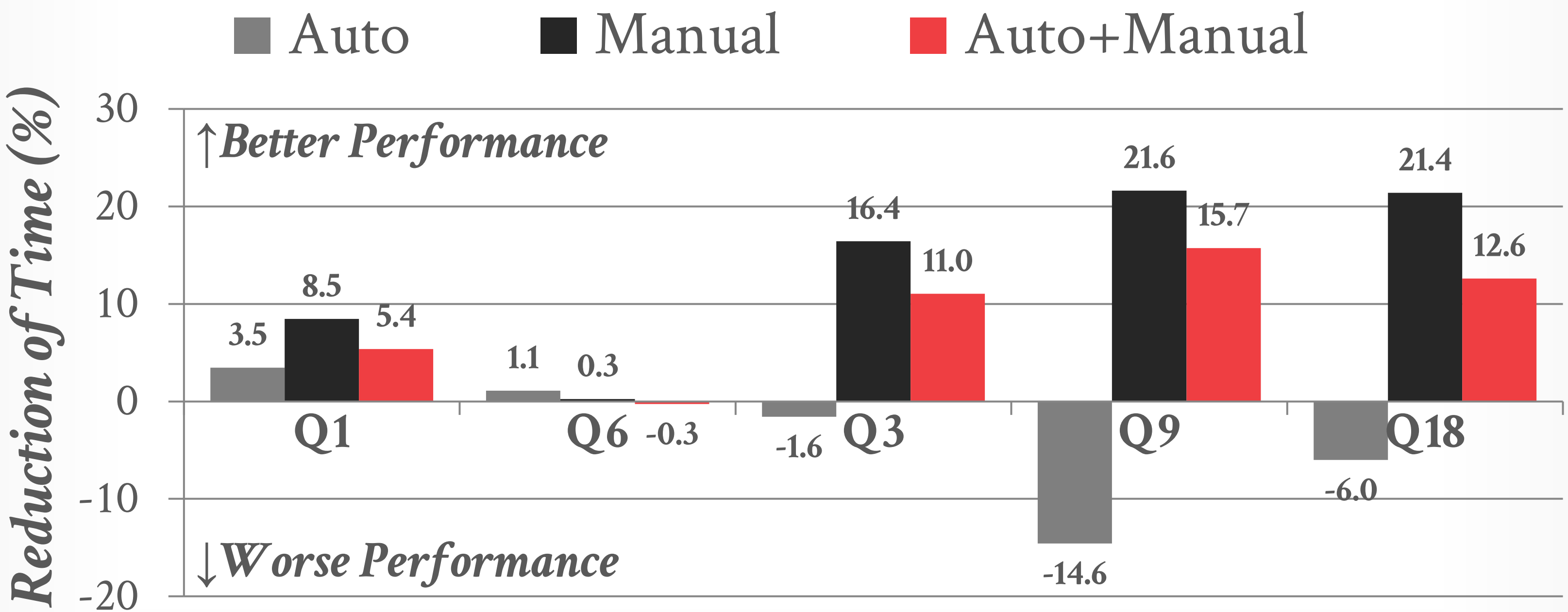
Intel Core i9-7900X (10 cores × 2HT)
Compiler: ICC v18



Source: Timo Kersten

AUTO-VECTORIZATION

Intel Core i9-7900X (10 cores × 2HT)
Compiler: ICC v18



Source: Timo Kersten

VECTORIZATION VS. COMPILATION

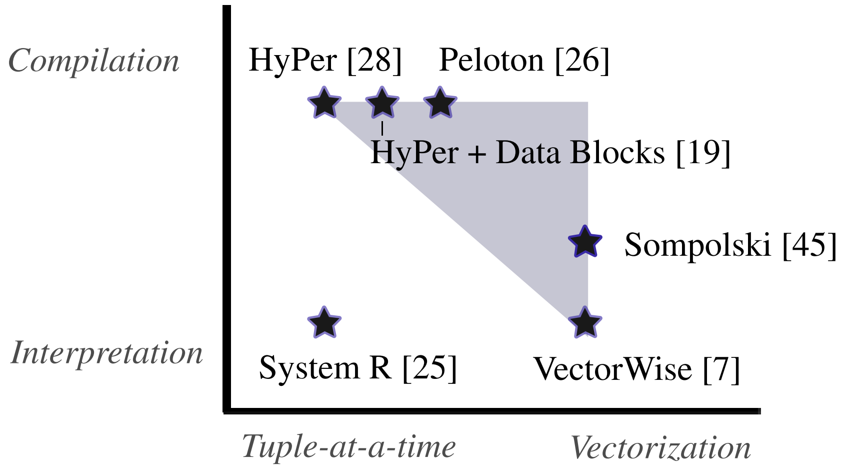


Figure 13: **Design space between vectorization and compilation** – *hybrid models integrate the advantages of the other approach.*

Source: Timo Kersten

PARTING THOUGHTS

No major performance difference between the Vectorwise and HyPer approaches for all queries.

NEXT CLASS

Hash Join Implementations

PROJECT #2 – DATABASE SYSTEM REPORT

Each student will write an encyclopedia article about the internals of a real-world DBMS.

→ We will target OLAP systems that implement topics from this semester.

Andy will post a sign-up sheet for you to pick what DBMS you want.

Feedback Due Date: April 1st

Final Due Date: May 1st

Prompt: *An aerial photograph of Carnegie Mellon University but with all the buildings replaced with giant database drums.*

https://15721.courses.cs.cmu.edu/spring2023/project2.html

DBDB.IO

Students will write their articles on CMU's online Database of Databases encyclopedia.

Each article will use a standard taxonomy.

→ For each feature category, you select pre-defined options for your DBMS.
→ You will then need to provide a summary paragraph with citations for that category.

DBDB.IO


Database of Databases
Browse
Leaderboards
Recent
Create
EmptyFields
Refine by
Search name, keywords, features...
AgensGraph
AlloyDB
AntDB
Mariposa
TransLattice
PostgreSQL
History
Checkpoints
Concurrency Control
Data Model
Foreign Keys
Website
Source Code
Tech Docs
Twitter
Developer
Country of Origin
Start Year
Former Name
Project Type
Written in
Operating Systems
Licenses
Wikipedia
Compatible Systems

☠ PLAGIARISM WARNING ☠

This article must be your own writing with your own images. You may **not** copy text/images directly from papers or other sources that you find on the web.

→ This includes both your submission for review and submission for your grade.

Plagiarism will **not** be tolerated.
See CMU's Policy on Academic Integrity for additional information.

PROJECT #3 – FINAL PROJECT

Group project to implement some substantial component or feature in a DBMS.

Projects should incorporate topics discussed in this course as well as from your own interests.

Each group must pick a project that is unique from their classmates.

***Prompt**: "A woman with a database drum for a head" by Johannes Vermeer*

https://15721.courses.cs.cmu.edu/spring2023/project3.html

PROJECT #3 – DELIVERABLES

Proposal Presentation: March 1st

Status Update Presentation: April 3rd

Design Document: TBA

Final Presentation: TBA

PROJECT #3 – PROPOSAL

Five-minute presentation to the class that discusses the high-level topic.

Each proposal must discuss:
→ Architecture and implementation overview of the project.
→ How you will test whether your implementation is correct.
→ What workloads you will use for your project.

PROJECT #3 – STATUS UPDATE

Five-minute presentation to update the class about the current status of your project.

Each presentation should include:

- → Current development status.
- → Whether your plan has changed and why.
- → Anything that surprised you during coding.

PROJECT #3 – DESIGN DOCUMENT

As part of the status update, you must provide a design document that describes your project implementation:

→ Architectural Design
→ Design Rationale
→ Testing Plan
→ Trade-offs and Potential Problems
→ Future Work

PROJECT #3 – FINAL PRESENTATION

10-minute presentation on the final status of your project during the scheduled final exam.

You should include any performance measurements or benchmarking numbers for your implementation.

Demos are always hot too…

PROJECT TOPICS

Fast Fixed-Point Decimals (Standalone)

Database Proxy Acceleration (PostgreSQL)

Adaptive Query Opt. (PostgreSQL)

UDF Inlining (DuckDB)

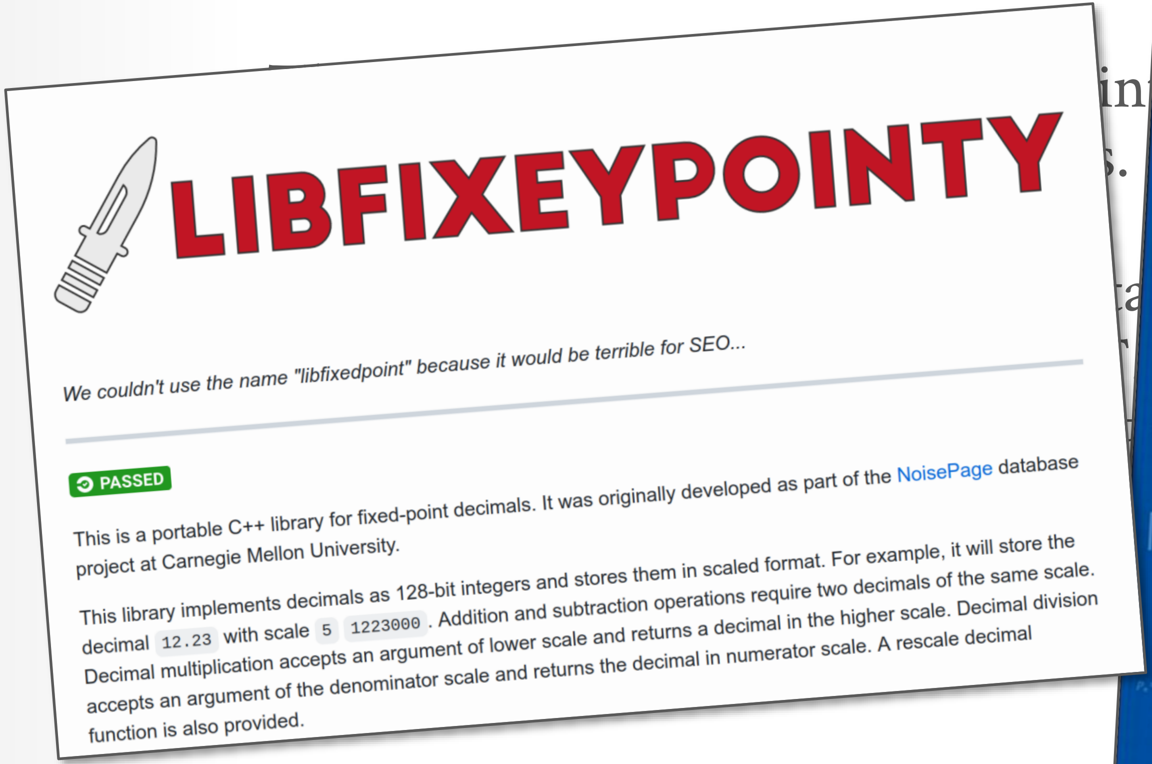
FAST FIXED-POINT DECIMALS

The Germans claim that fixed-point decimals are faster than floating point decimals.

Project: Complete our implementation and integrate into PostgreSQL as UDT.

→ Verify that the calculations are correct.
→ Add support for vertical vectorization.
→ Benchmark against PostgreSQL's built-in `NUMERIC`.

FAST FIXED-POINT D



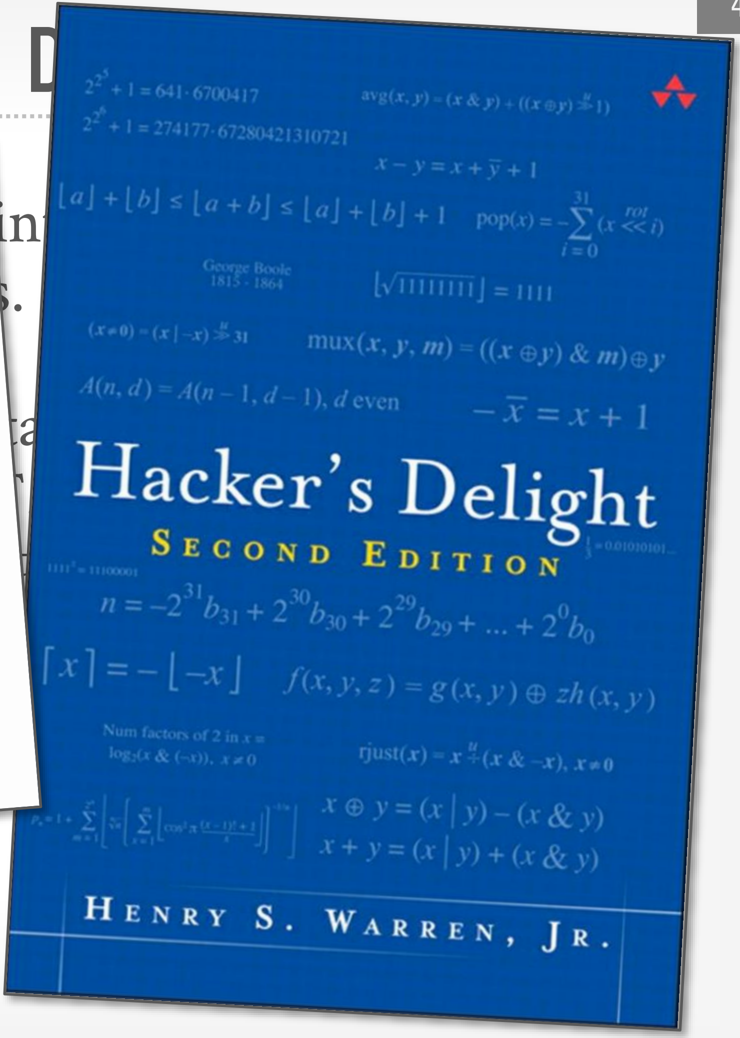
LIBFIXEYPOINTY

We couldn't use the name "libfixedpoint" because it would be terrible for SEO...

PASSED

This is a portable C++ library for fixed-point decimals. It was originally developed as part of the NoisePage database project at Carnegie Mellon University.

This library implements decimals as 128-bit integers and stores them in scaled format. For example, it will store the decimal `12.23` with scale `5` `1223000`. Addition and subtraction operations require two decimals of the same scale. Decimal multiplication accepts an argument of lower scale and returns a decimal in the higher scale. Decimal division accepts an argument of the denominator scale and returns the decimal in numerator scale. A rescale decimal function is also provided.



DATABASE PROXY ACCELERATION

We have been working on optimizing network operations in PostgreSQL proxies.

Project: Extend pgCat (Rust) to support user-bypass and/or kernel-bypass.

→ User-bypass: eBPF
→ Kernel-bypass: io_uring
→ Matt has existing benchmark scripts to compare against his proxy and Odyssey.

ADAPTIVE QUERY OPTIMIZATION

We want to be able to change a query plan during execution without stopping the query.

Project: Create a PostgreSQL extension that swaps a plan node in the tree with a "dummy" node.

→ New node can either halt execution or generate fake data.
→ An easier approach might be to wrap nodes with "control" nodes that determine whether to call inner node.

UDF INLINING

We want to compare methods for compiling UDFs into machine code versus UDF inlining.

→ We will cover this in Lecture #14

Project: Add support for PL/pgSQL UDF inlining in DuckDB.

→ PostgreSQL's query optimizer is too primitive.

→ DuckDB supports nested query decorrelation, which is needed for the Microsoft Froid technique.

→ Potentially in collaboration with (different) Germans.

HOW TO START

Form a team. Sign-up on class spreadsheet.

Meet with your team and discuss potential topics.

Look over source code and determine what you will need to implement.

I am able during Spring Break for additional discussion and clarification of the project idea.