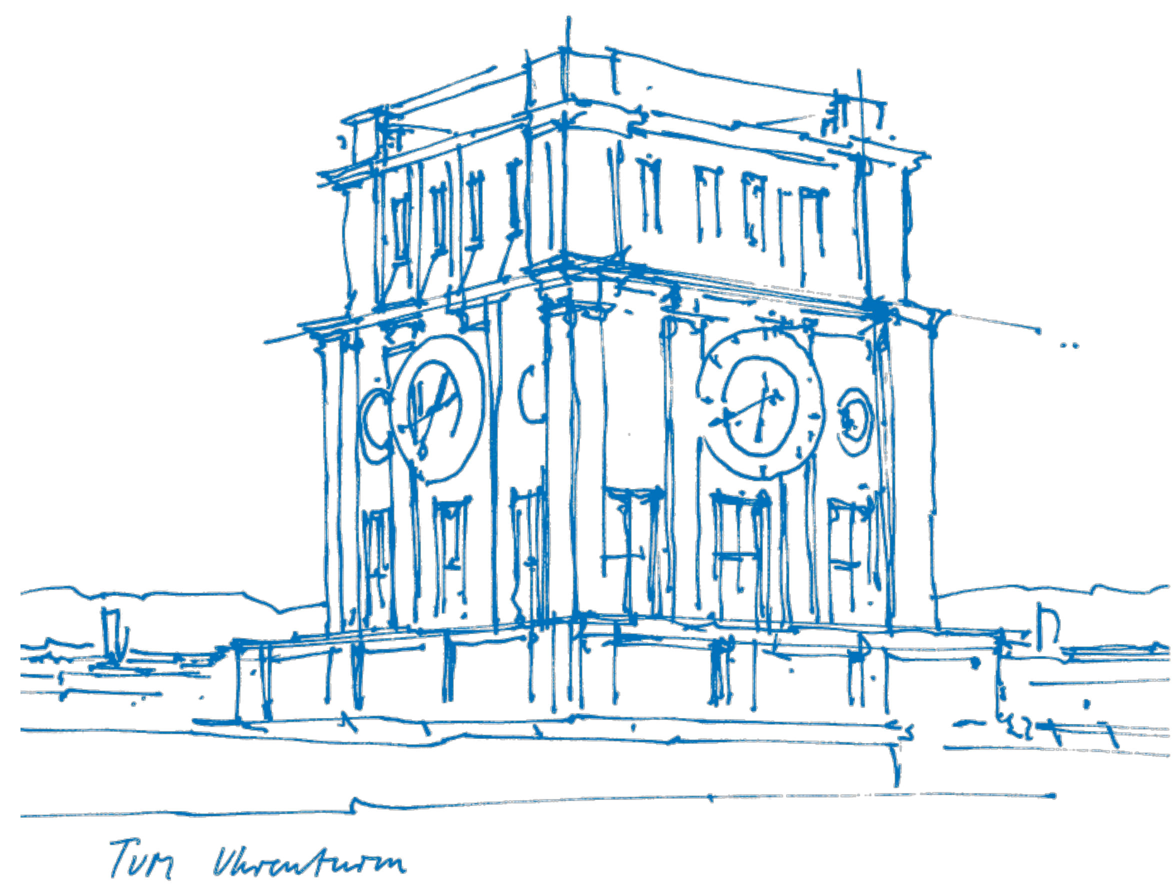
Bonusproject 2, Execution Engines

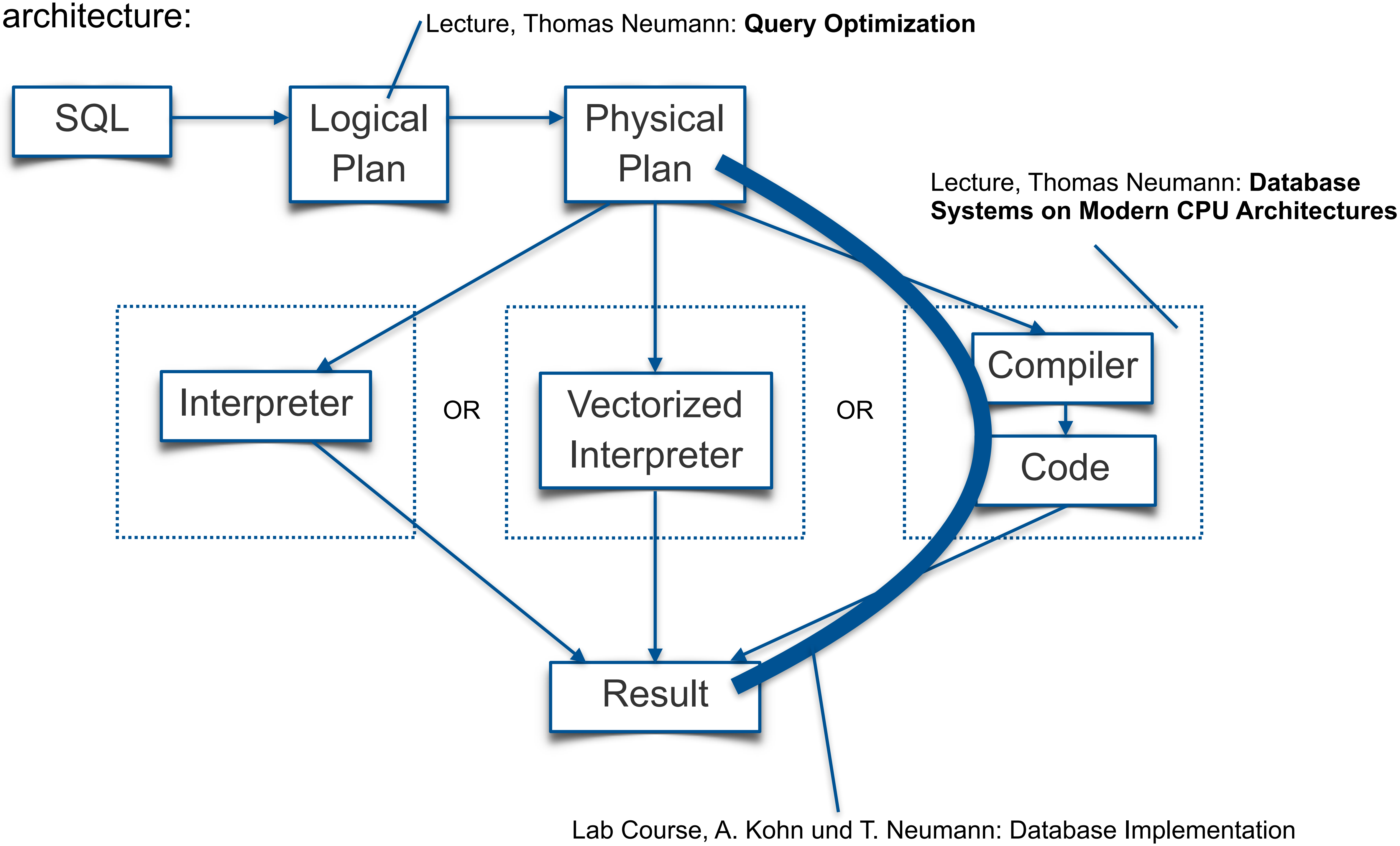
Timo Kersten
Technische Universität München
Faculty for Computer Science
Chair for Database Systems



DBMS Execution Engines

Execution Engines

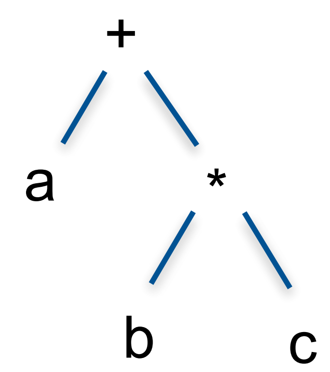
DBMS architecture:



Traditional Interpreters (in DBMS)

Volcano style interpreters
G. Graefe, Volcano - An Extensible and Parallel Query Evaluation System

Example: Expression evaluator: a + b * c



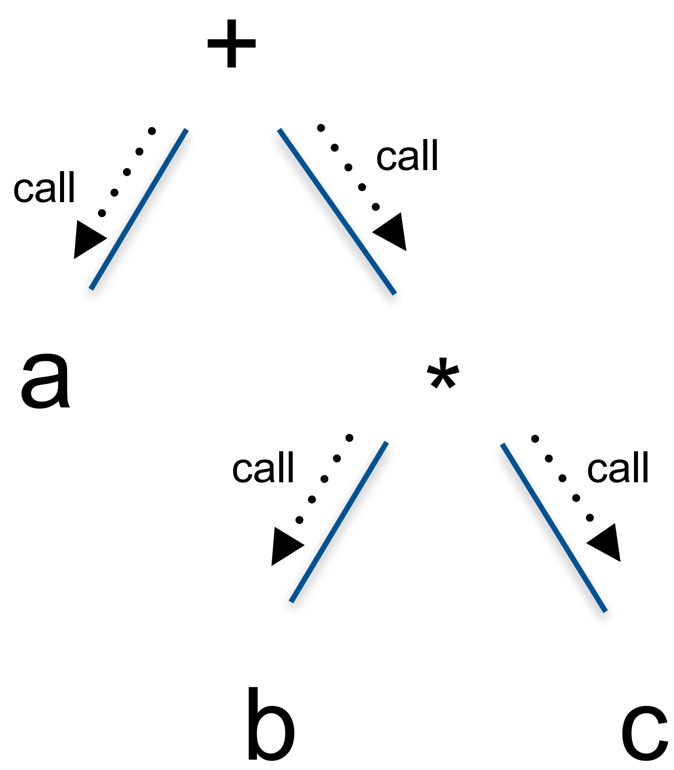

```
class Operator;
class BinaryOperator;
class Plus;
class Mul;

class Operator {
  public:
   virtual int compute();
};

class BinaryOperator : public Operator {
  protected:
   Operator *left, *right;
};

class Plus : public BinaryOperator {
   int compute() override {
      auto l = left->compute();
      auto r = right->compute();
      return l + r;
   }
};
```

Runtime Behavior



Many virtual function calls
➡ Register value saving, memory traffic
➡ Hard to predict branches, miss penalty 15c
➡ Extra instructions

Vectorized Interpreter

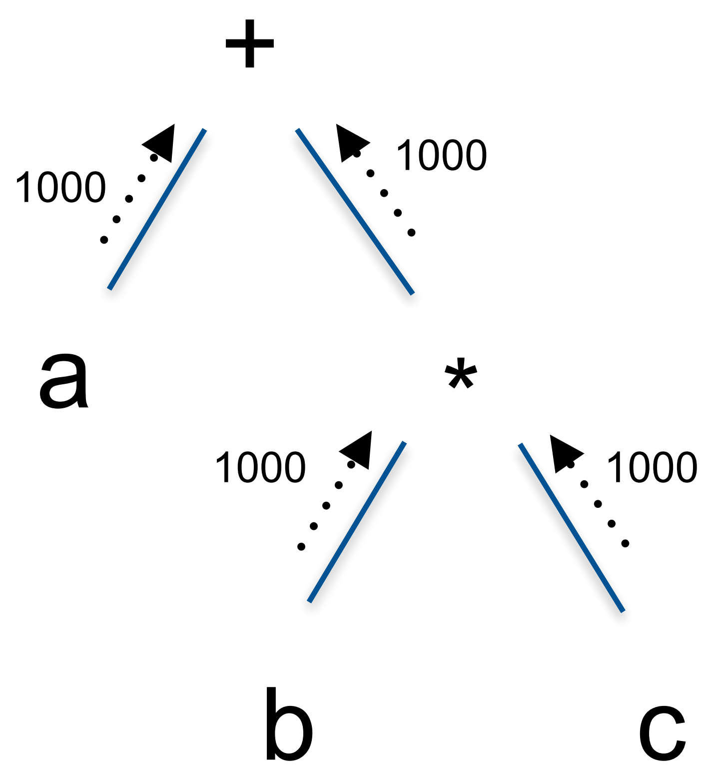
Idea: Penalties are on a per call basis. Let's pass multiple elements per call
-> Amortize call cost over batch

```
#include <vector>

class Operator;
class BinaryOperator;
class Plus;

class Operator {
  public:
   virtual vector<int> compute();
};
…
class Plus : public BinaryOperator {
   vector<int> compute() override {
      auto l = left->compute();      One call per batch
      auto r = right->compute();
      vector<int> result;
      for (int i = 0; i < l.size(); ++i)      Tight loop over elements
       result.push_back(l[i] + r[i]);
      return result;
   }
};
```

Vectorized Interpreter Runtime Behavior



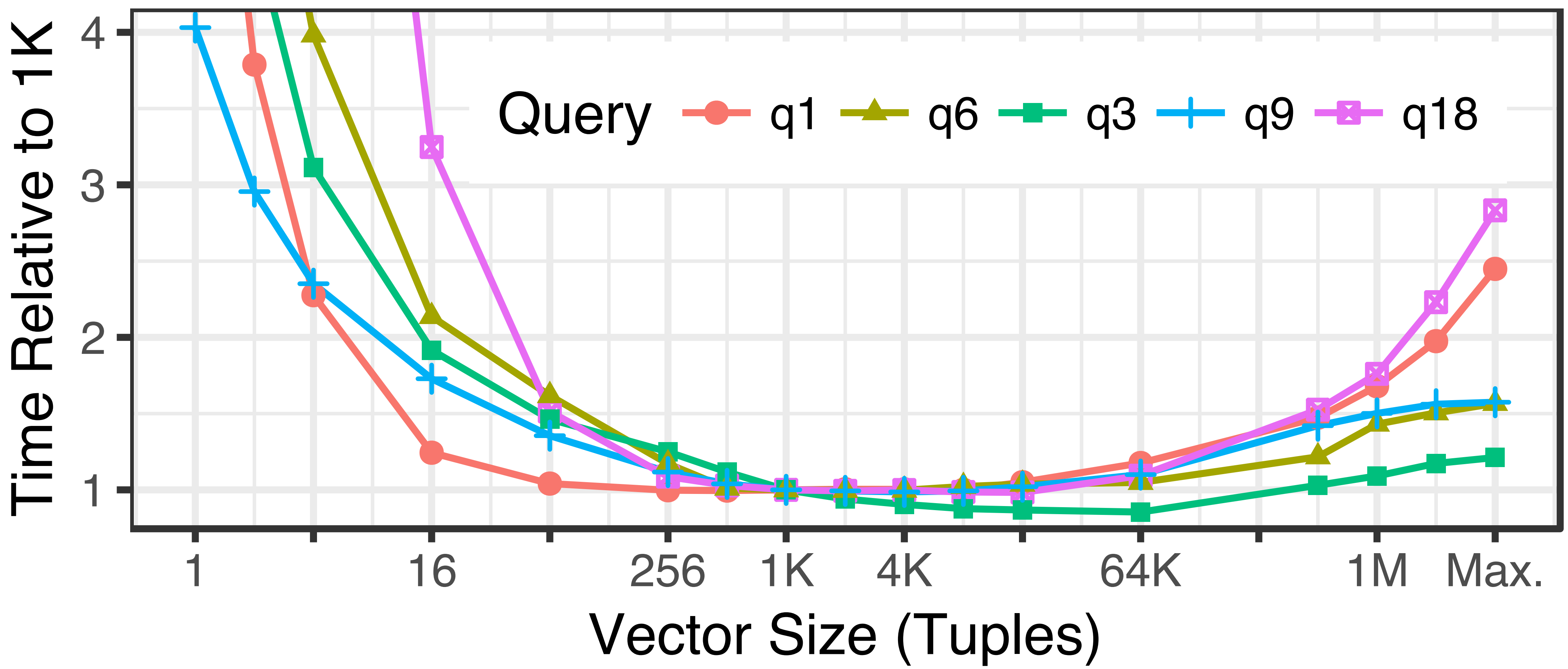
Amortizes call overhead.

Complications:

- Memory traffic
- Control flow, e.g. selection, join, grouping
  ➡ Selection vectors
  ➡ Sparsely populated vectors after operation, less effective amortisation
- Type combinations, e.g. combined hash table keys

Optimal Vector Size

Runtimes of selected TPC-H queries
Sf=1



Trade-off between amortisation and cache capacity

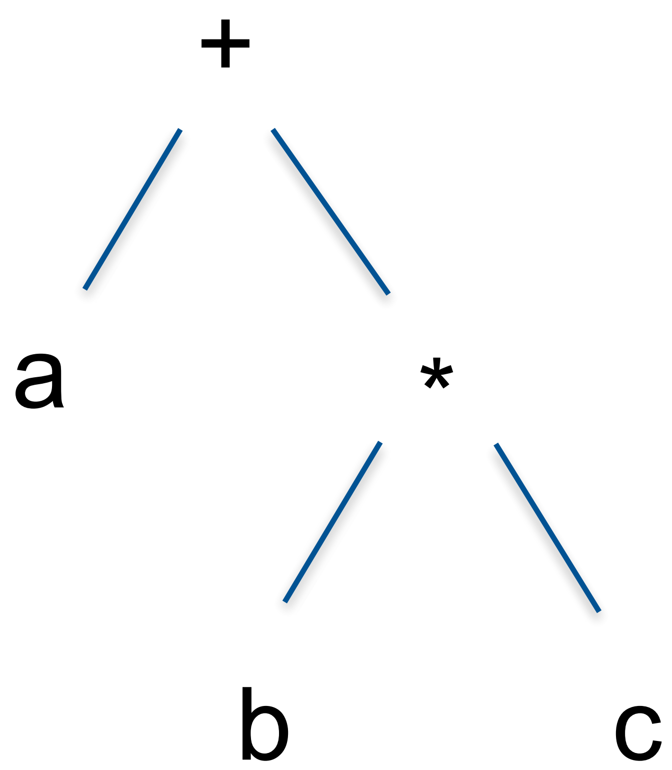
Vectorized Interpreter: Pros

Remember: Some restrictions on type combinations

But some advantages:

- Call overhead is amortised
- Each primitive can use the full power of C to work on multiple elements
  - That means we can easily use hardware features, e.g. SIMD, hashing instructions, prefetching.
- Everything precompiled: Query execution can start right away.

Code Compilation




```
int compiledFun(int a, int b, int c) {
  return a + b * c;
}
```

Pro:

- No virtual function calls
- Intermediate values can be kept in registers
- Compiler can choose minimal number of instructions
- Data flow becomes control flow

Cons:

- Compile time
- How to use SIMD etc.?

If one wants to build a new query engine today, which paradigm should one use?

Vectorwise vs. HyPer?

actian
Vectorwise

HyPer

MonetDB/X100: Hyper-Pipelining Query Execution, Boncz et al., CIDR 2005

Other systems with vectorized architecuture:
Quickstep, Snowflake, …

Efficiently Compiling Efficient Query Plans for Modern Hardware, Neumann, VLDB 2011

Other systems with query compilation:
Peloton, Microsoft Hekaton, Spark, …

Not directly comparable:

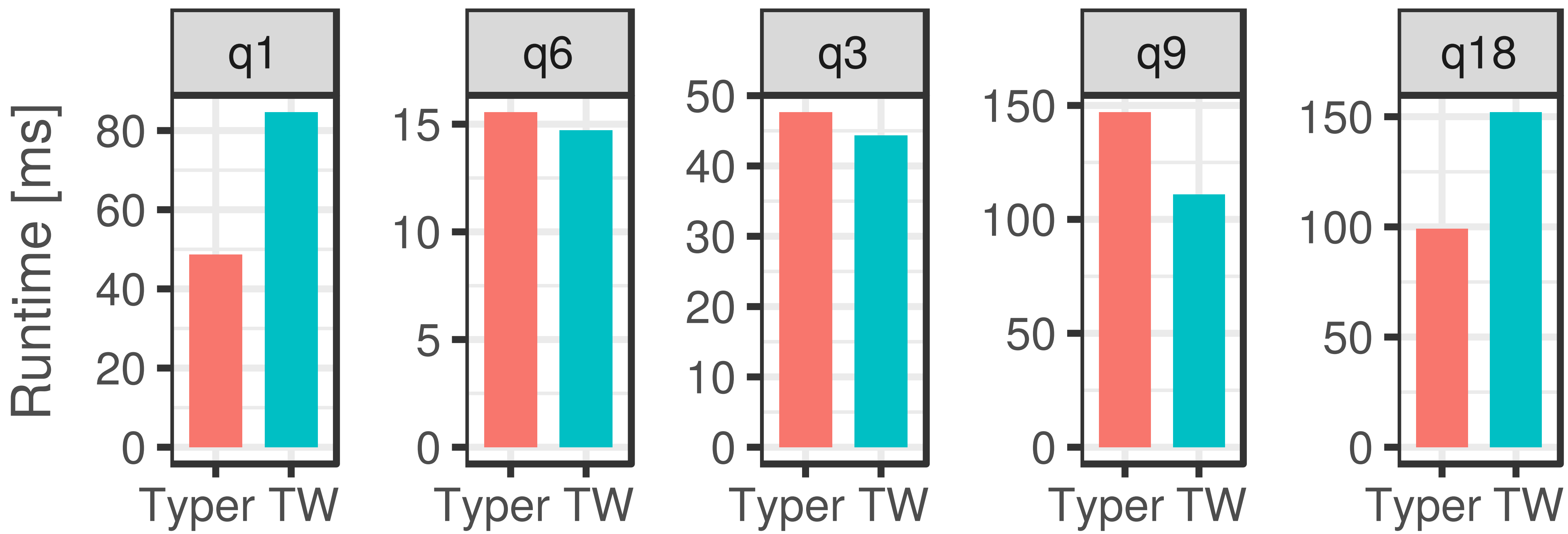
- Different storage/compression schemes
- Different query processing algorithms and data structure
- Different parallelisation frameworks
- Different query optimisers
- …

Tectorwise vs. Typer

Back to back comparison: Implementation of both paradigms

- Same storage/compression schemes (mmap-ed uncompressed columns)
- Same query processing algorithms and data structure (down to identical hash tables)
- Same parallelisation frameworks (morsel-driven parallelization)
- Same query optimisers (identical query plans)
- …

TPC-H (SF=1, 1 thread, no SIMD)



Main cost from:

q1: Expression Evaluation
q6: Selection
q3: Join with small hts
q9: Join with big hts
q18: Grouping

Microarchitectural Analysis: TPC-H q1

```sql
select   l_returnflag, l_linestatus,
         sum(l_quantity),
         sum(l_extendedprice),
         sum(l_extendedprice * (1 - l_discount)),
         sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
         avg(l_quantity),
         avg(l_extendedprice),
         avg(l_discount),
         count(*)
from     lineitem
where    l_shipdate <= date '1998-12-01' - interval '90' day
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus
```

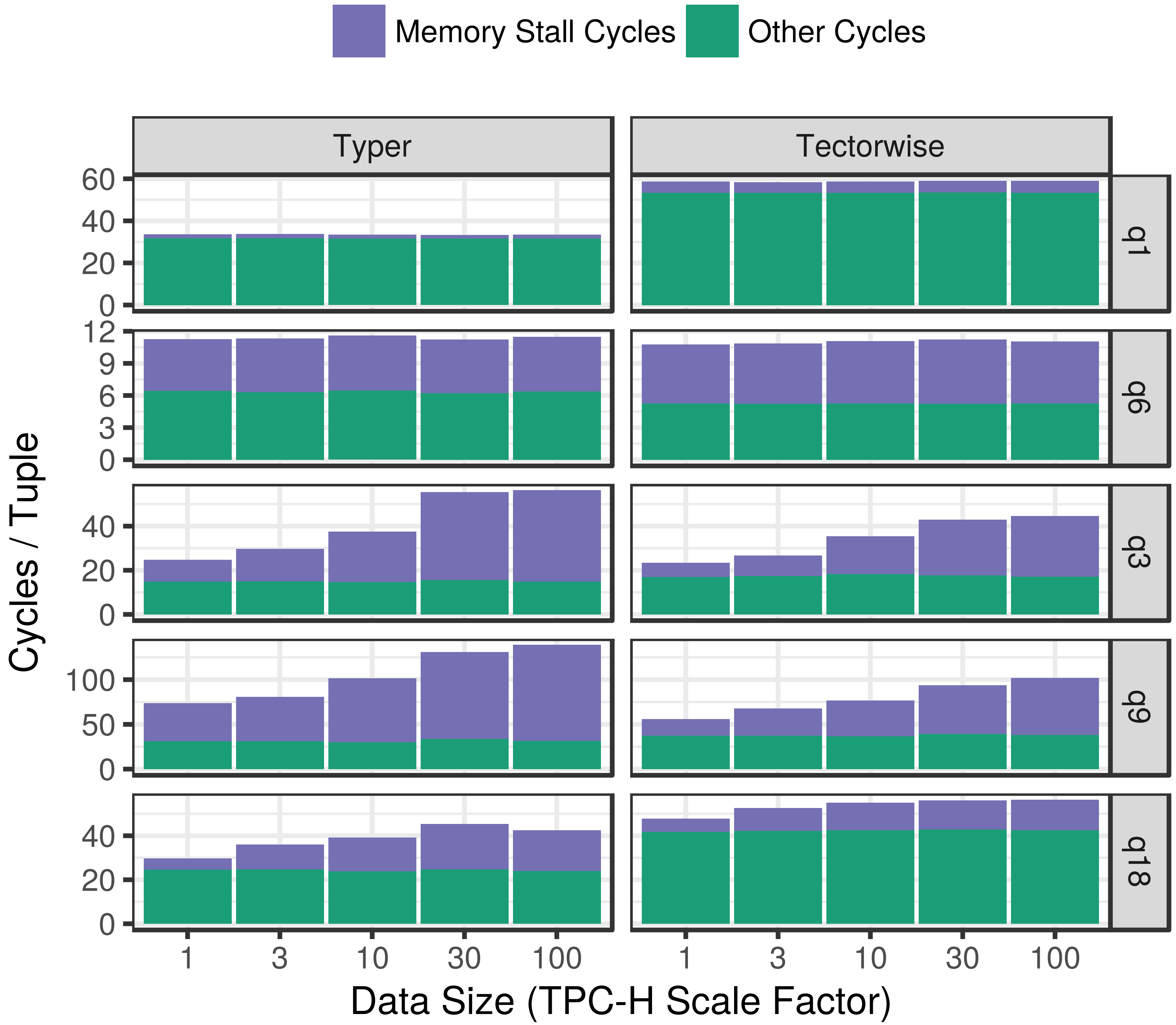
	cycles	IPC	instr.	L1 miss	LLC miss	branch miss	mem. stall cycles
Typer	34	2.0	**68**	0.6	0.57	0.01	1.8
Tectorwise	59	2.8	**162**	2.0	0.57	0.03	5.2

Microarchitectural Analysis: TPC-H q9

```sql
select ...
from lineitem, orders, partsupp,
     supplier, part, nation
where s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%green%'
```

	cycles	IPC	instr.	L1 miss	LLC miss	branch miss	mem. stall cycles
Typer	74	0.6	42	1.7	0.46	0.34	**42**
Tectorwise	56	1.3	76	2.1	0.47	0.39	**18**

Stall Cycles



OLAP Throughput

< *Computation*: compiled code can keep data in CPU registers

> *Parallel data access*: vectorisation is better at generating parallel cache misses

= *SIMD*: easier to apply, though gains are limited

= *Parallelization*: both scale well

= *Hardware platforms*: Skylake, Knights Landing, and AMD Ryzen are similar

➡ overall performance similar on OLAP, no clear winner

Other Aspects Are Probably More Important

< *OLTP*: compilation results in fast stored procedures

< *Language support*: compilation enables seamless integration of different languages

> *Compile time*: vectorisation has no JIT-compilation overhead

> *Profiling*: Runtime can easily be attributed to vectorised primitives

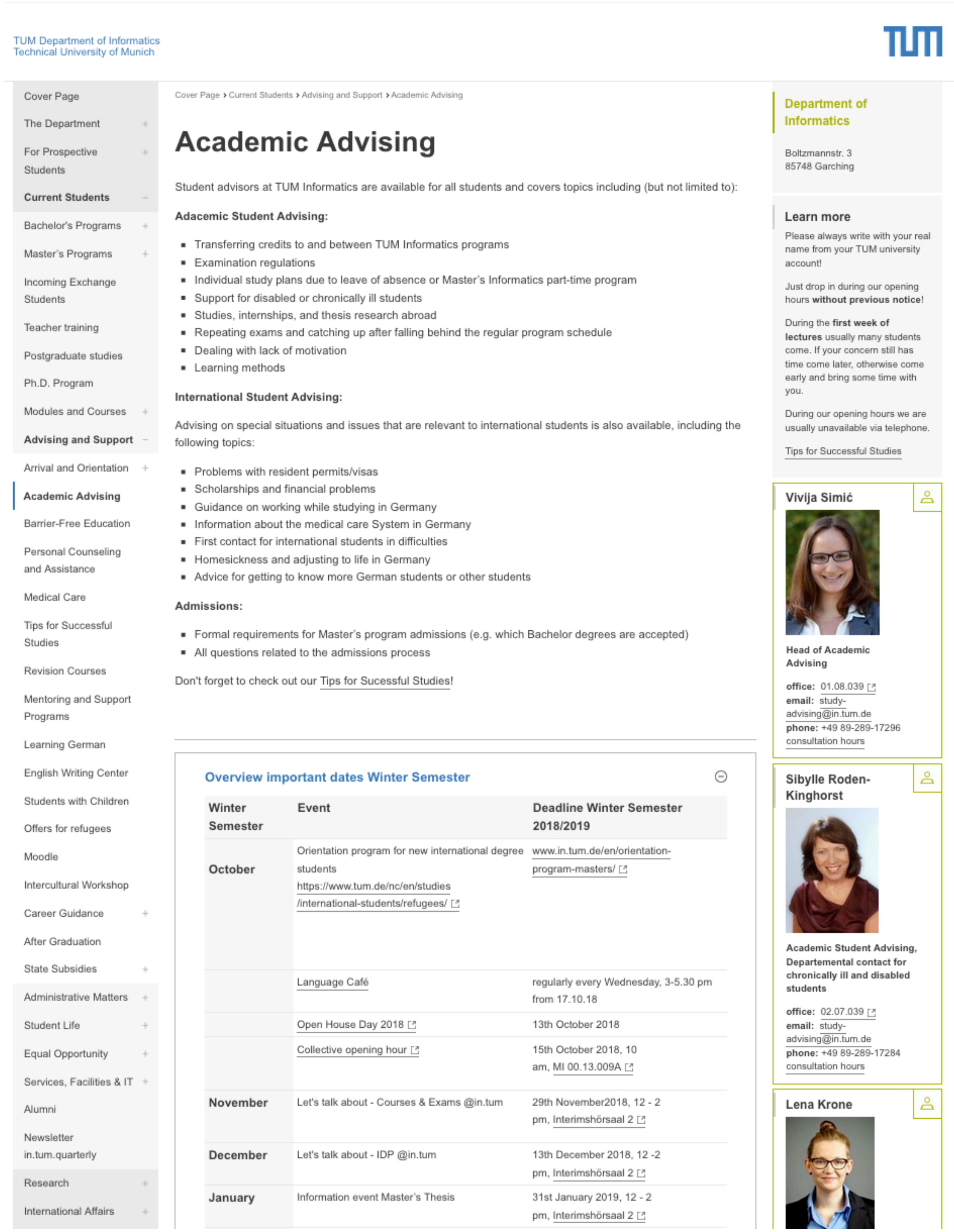
> *(Micro-)adaptivity*: vectorisation allows primitives to be swapped mid-flight

for optimal performance, one needs a hybrid engine that combines both approaches

If You Are New to TUM

www.in.tum.de/academic-advising

Have a look at the “Let’s talk about …” series.



References

Z-order curve: https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/?sc_channel=sm&sc_campaign=zackblog&sc_country=global&sc_geo=global&sc_category=rds&sc_outcome=aware&adbsc=awsdbblog_social_20170517_72417147&adbid=864895517733470208&adbpl=tw&adbpr=66780587

Bloom Filter https://www.kdnuggets.com/2016/08/gentle-introduction-bloom-filter.html

Execution Engine Comparison
Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask, Kersten et al., PVLDB 2018