Lecture #16

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Vectorization vs. Compilation

@Andy_Pavlo // 15-721 // Spring 2020

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

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

VECTORWISE – PRECOMPILED PRIMITIVES

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

σ str_col='abc' && int_col=4

↑

foo

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

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

HYPER – JIT QUERY COMPILATION

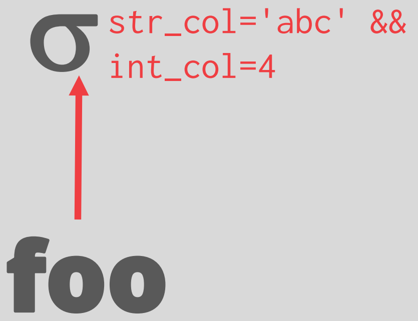
Compile queries in-memory into native code using the LLVM toolkit.

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 – JIT QUERY COMPILATION

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




```
vec<offset> sel_eq_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

Relaxed Operator Fusion

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

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

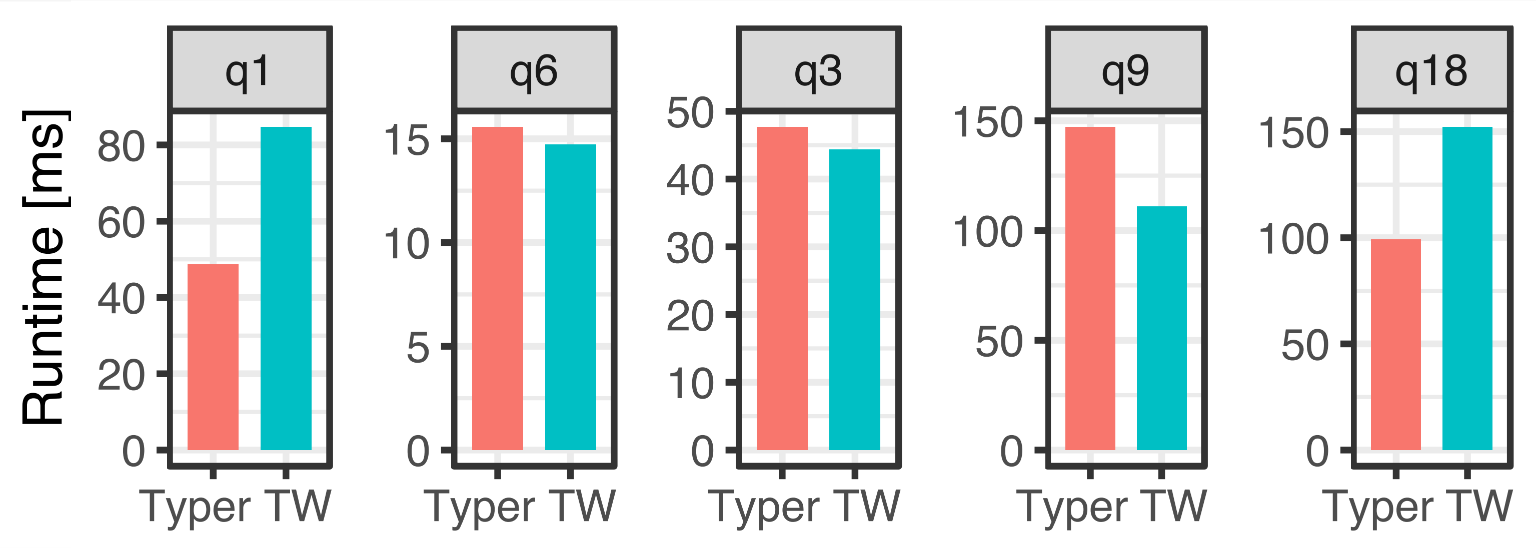
Q3: Join (build: 147k tuples / probe: 3.2m tuples)

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

SINGLE-THREADED PERFORMANCE



Source: Timo Kersten

SINGLE-THREADED PERFORMANCE

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

MAIN FINDINGS

Both models are efficient and achieve roughly the same performance.

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

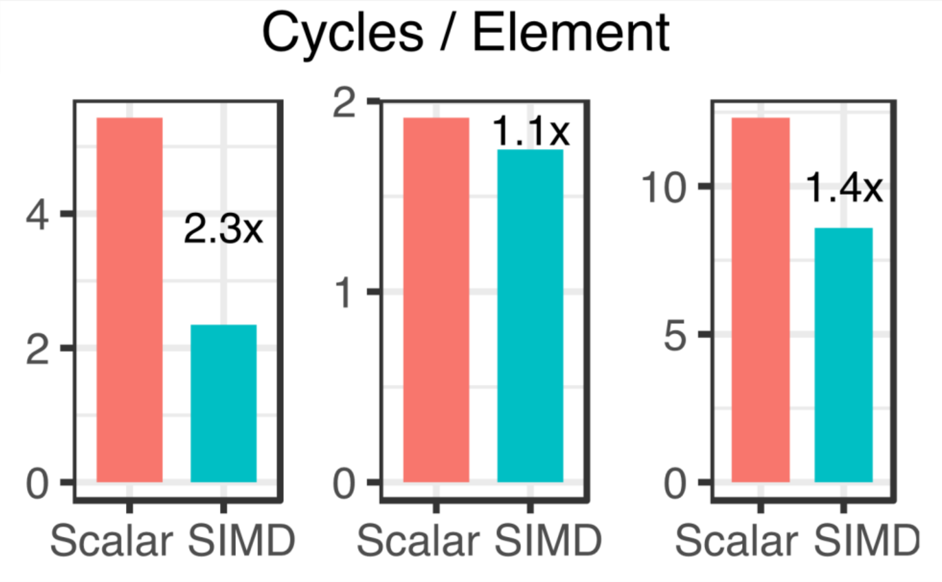
Vectorization is slightly better at hiding cache miss latencies.

SIMD PERFORMANCE

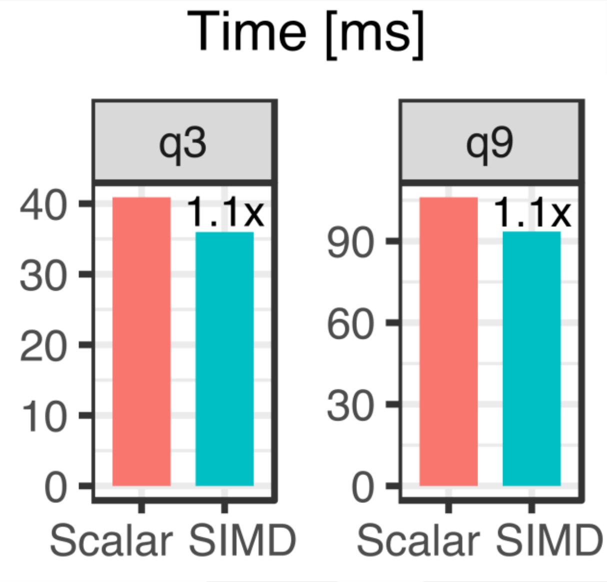
Evaluate vectorized branchless selection and hash probe in Tectorwise.

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

SIMD EVALUATION



Hashing *Gather* *Join*



Source: Timo Kersten

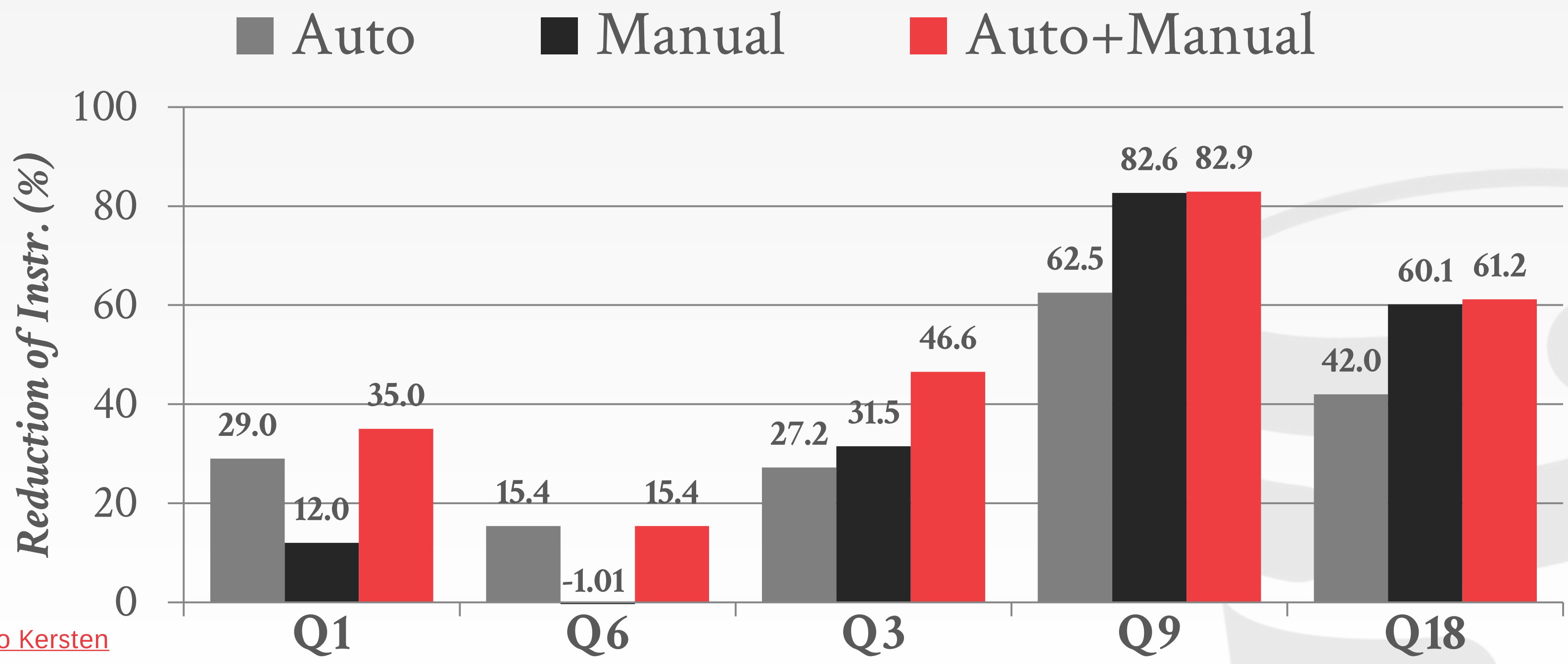
AUTO-VECTORIZATION

Measure 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

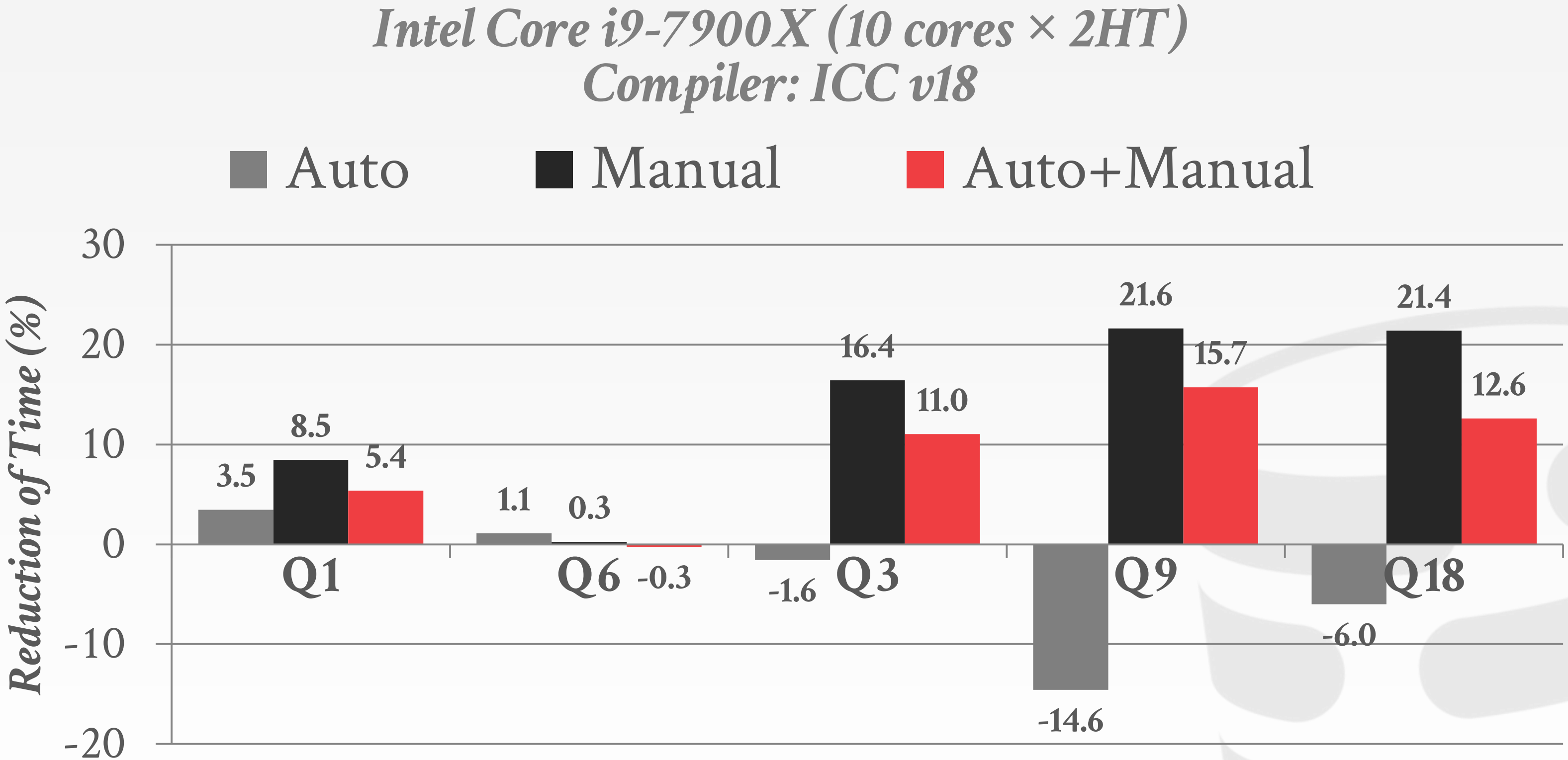
AUTO-VECTORIZATION

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



Source: Timo Kersten

AUTO-VECTORIZATION



Source: Timo Kersten

OBSERVATION

The paper (partially) assumes that vectorization and compilation are mutually exclusive.

HyPer fuses operators together so that they work on a single tuple a time to maximize CPU register reuse and minimize cache misses.

VECTORIZATION VS. COMPILATION

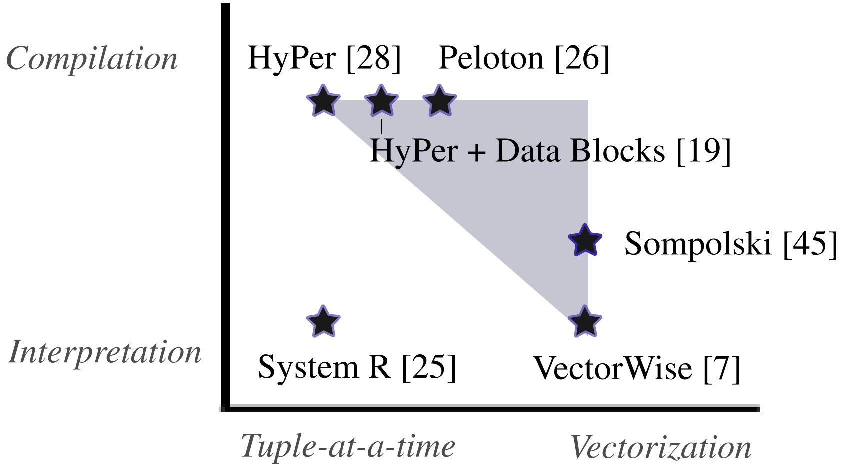


Figure 13: **Design space between vectorization and compilation** – *hybrid models integrate the advantages of the other approach.*

Source: Timo Kersten

PIPELINE PERSPECTIVE

Each pipeline **fuses** operators together into loop

Each pipeline is a **tuple-at-a-time** process

PIPELINE PERSPECTIVE

Each pipeline **fuses** operators together into loop

Each pipeline is a **tuple-at-a-time** process

Emit

↑

Agg

↑

Filter

↑

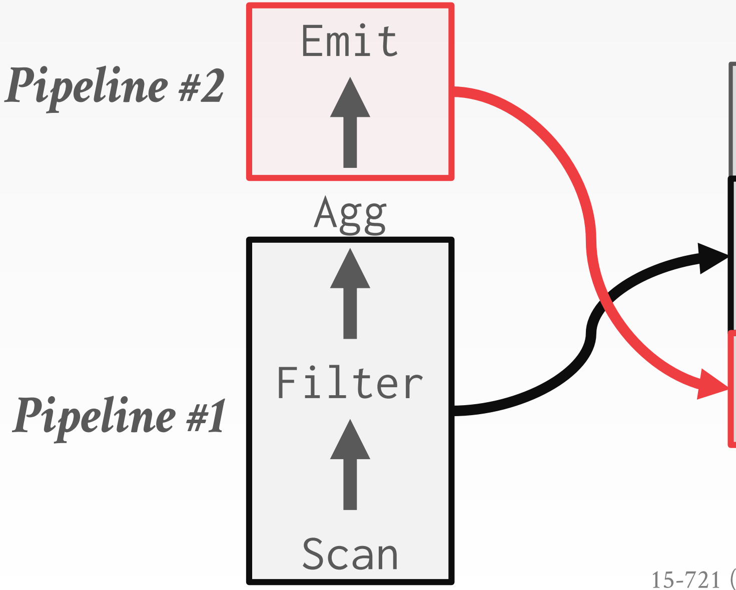
Scan

```
def plan(state):
  agg = dict()
  for t in A:
    if t.age > 20:
      agg[t.city]['count']++
  for t in agg:
    emit(t)
```

PIPELINE PERSPECTIVE

Each pipeline **fuses** operators together into loop

Each pipeline is a **tuple-at-a-time** process




```
def plan(state):
  agg = dict()
  for t in A:
    if t.age > 20:
      agg[t.city]['count']++
  for t in agg:
    emit(t)
```

FUSION PROBLEMS

Fusion inhibits some optimizations:

→ Unable to look ahead in tuple stream.

→ Unable to overlap computation and memory access.

Scan → `for t in A:`
Filter → `if t.age > 20:`
Agg → `agg[t.city]['count']++`

```
def plan(state):
  agg = dict()
  for t in A:
    if t.age > 20:
      agg[t.city]['count']++
  for t in agg:
    emit(t)
```

FUSION PROBLEMS

Fusion inhibits some optimizations:
→ Unable to look ahead in tuple stream.
→ Unable to overlap computation and memory access.

Cannot SIMD

Scan → Filter → Agg →

```
def plan(state):
  agg = dict()
  for t in A:
    if t.age > 20:
      agg[t.city]['count']++
  for t in agg:
    emit(t)
```

FUSION PROBLEMS

Fusion inhibits some optimizations:

→ Unable to look ahead in tuple stream.

→ Unable to overlap computation and memory access.

☠ *Cannot SIMD*

☠ *Cannot Prefetch*

Scan → *Filter* → *Agg* →

```
def plan(state):
  agg = dict()
  for t in A:
    if t.age > 20:
      agg[t.city]['count']++
  for t in agg:
    emit(t)
```

RELAXED OPERATOR FUSION

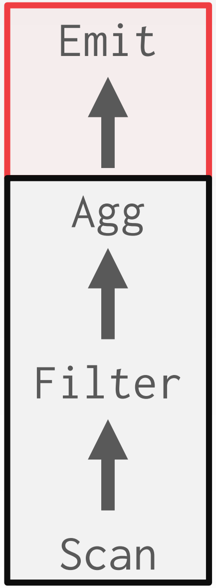
Vectorized processing model designed for query compilation execution engines.

Decompose pipelines into **stages** that operate on vectors of tuples.

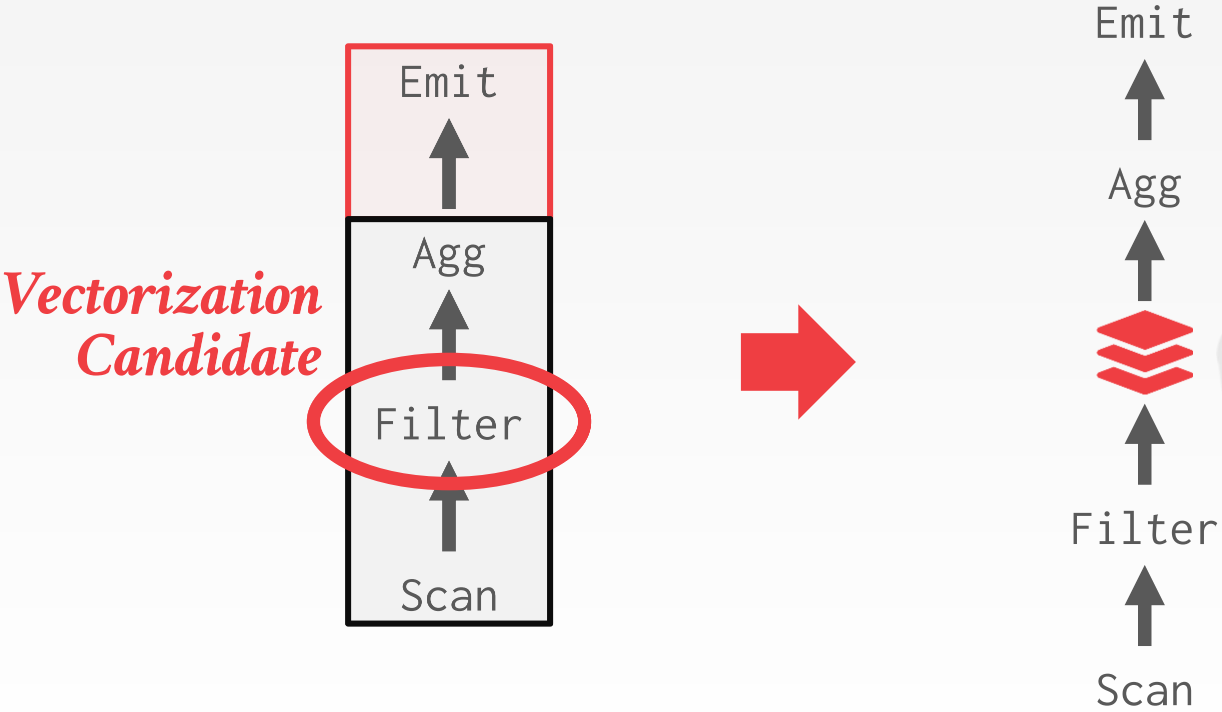
→ Each stage may contain multiple operators.
→ Communicate through cache-resident buffers.
→ Stages are granularity of vectorization + fusion.

RELAXED OPERATOR FUSION FOR IN-MEMORY DATABASES: MAKING COMPILATION, VECTORIZATION, AND PREFETCHING WORK TOGETHER AT LAST
VLDB 2017

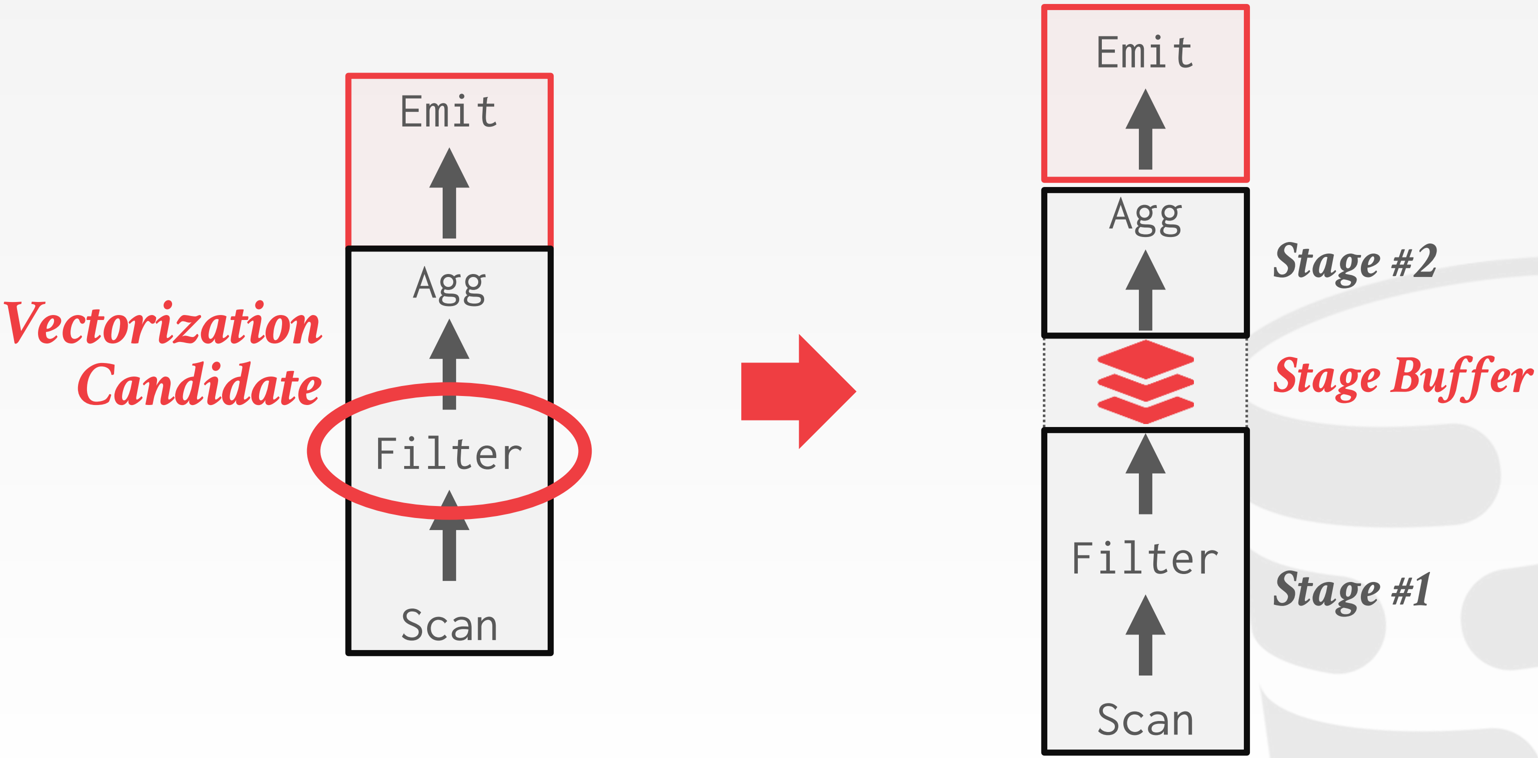
ROF EXAMPLE



ROF EXAMPLE


Vectorization Candidate
Emit
Agg
Filter
Scan
Emit
Agg
Filter
Scan

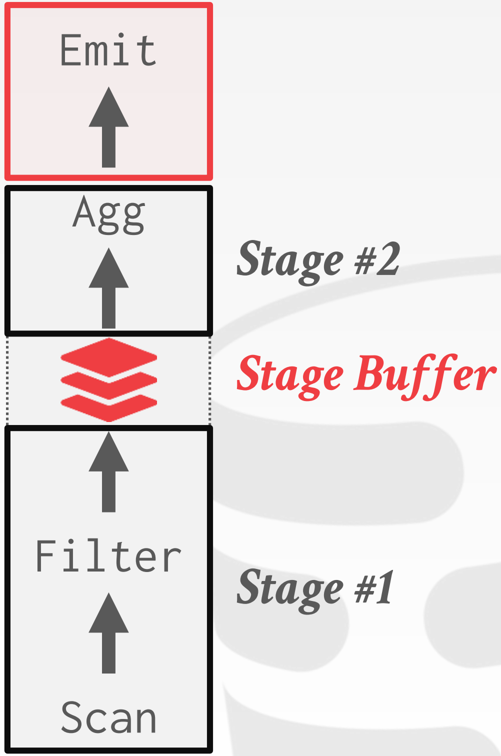
ROF EXAMPLE



Emit
Agg
Filter
Scan
Vectorization Candidate
Emit
Agg
Stage #2
Stage Buffer
Filter
Scan
Stage #1

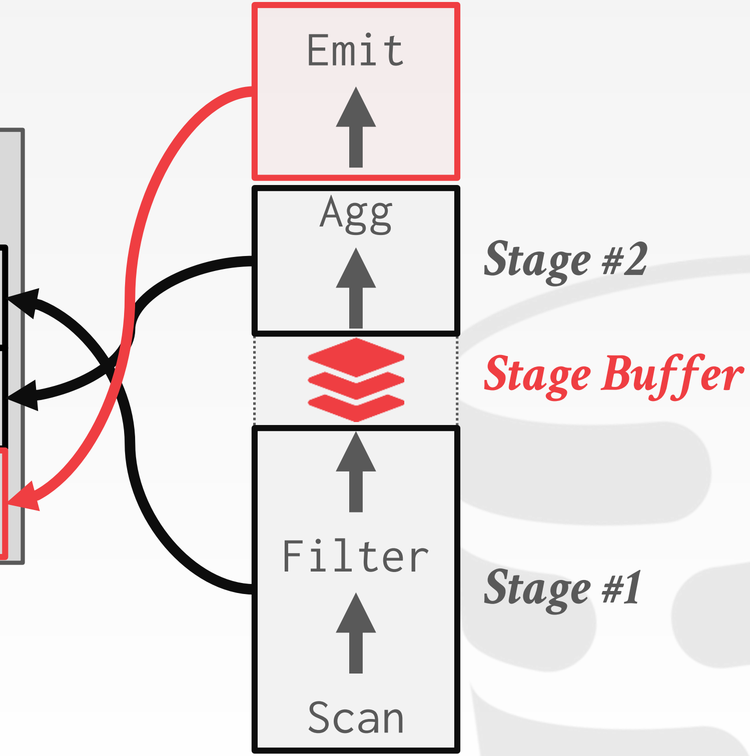
ROF EXAMPLE

```
def plan(state):
  agg = dict()
  for t in A step 1024:
    out = simd_cmp_gt(t, 20, 1024)
    for ft in out:
      agg[ft.city]['count']++
  for t in agg:
    emit(t)
```



ROF EXAMPLE

```
def plan(state):
  agg = dict()
  for t in A step 1024:
    out = simd_cmp_gt(t, 20, 1024)
    for ft in out:
      agg[ft.city]['count']++
  for t in agg:
    emit(t)
```



ROF SOFTWARE PREFETCHING

The DBMS can tell the CPU to grab the next vector while it works on the current batch.

→ Prefetch-enabled operators define start of new stage.

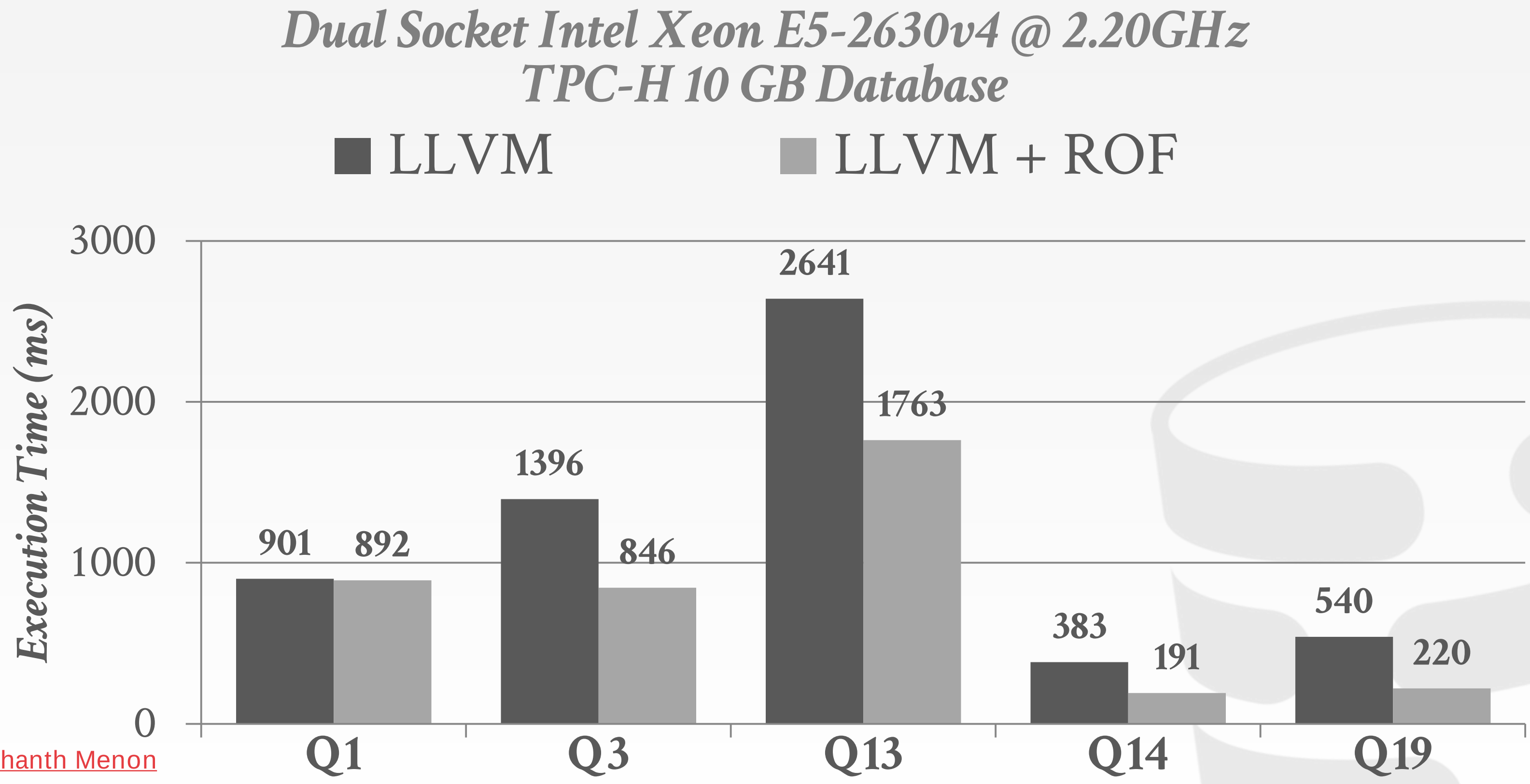
→ Hides the cache miss latency.

Any prefetching technique is suitable

→ Group prefetching, software pipelining, AMAC.

→ Group prefetching works and is simple to implement.

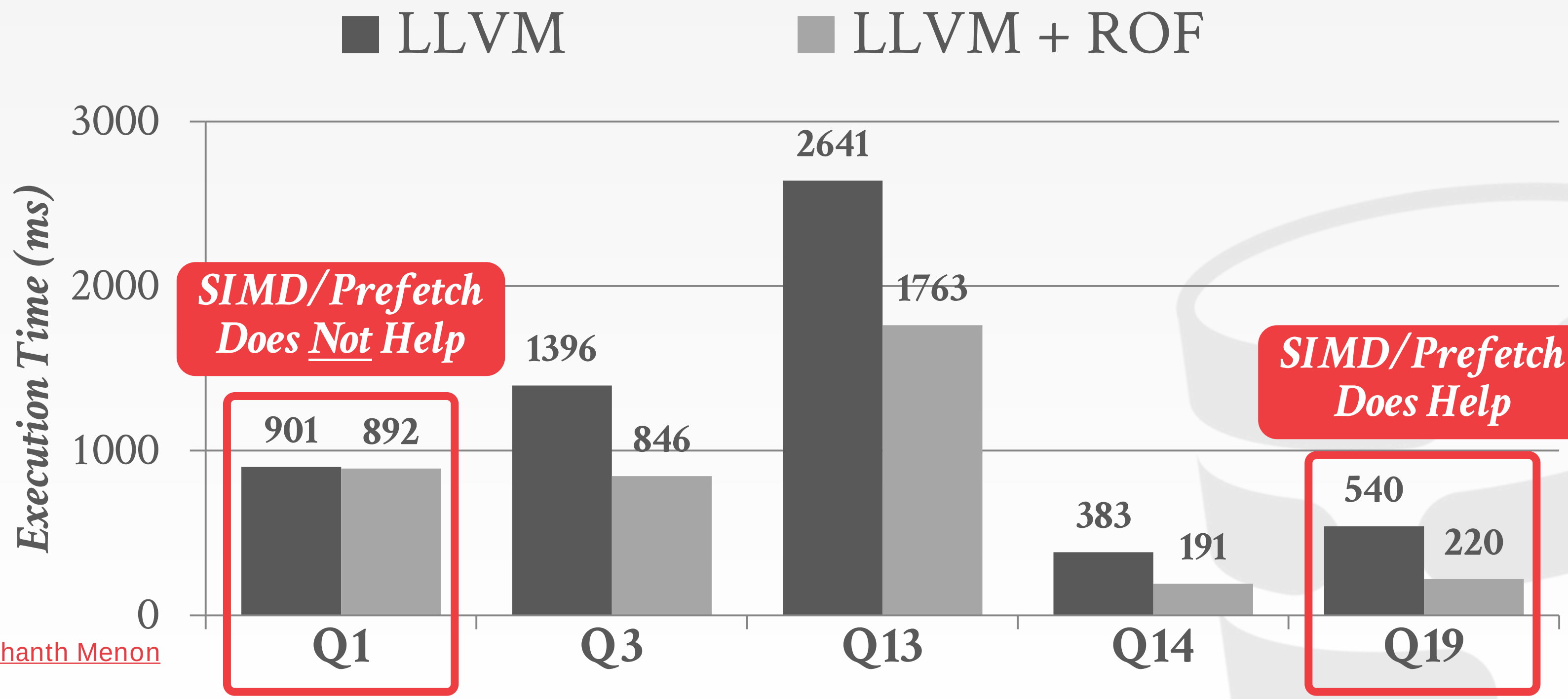
ROF EVALUATION



Source: Prashanth Menon

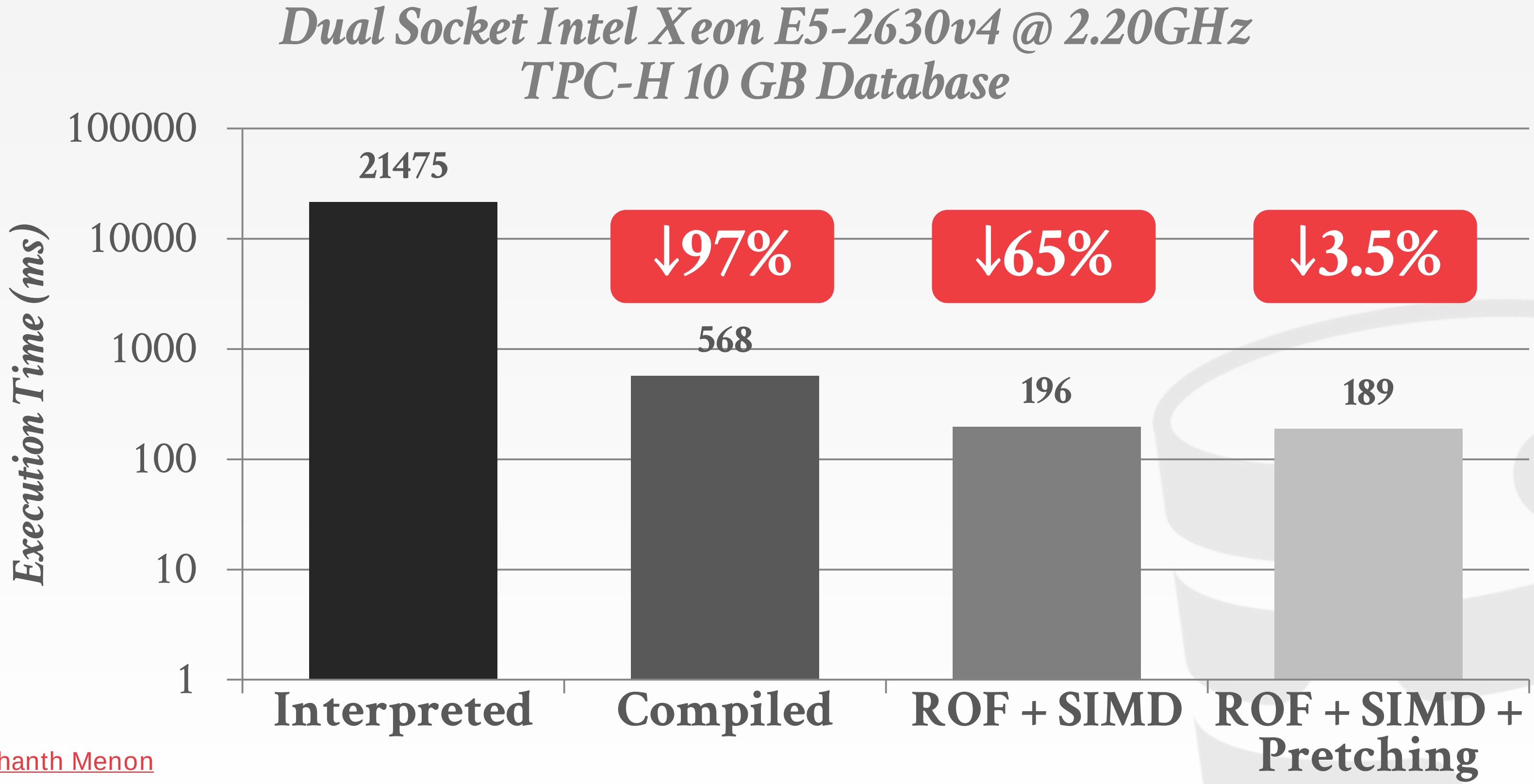
ROF EVALUATION

Dual Socket Intel Xeon E5-2630v4 @ 2.20GHz
TPC-H 10 GB Database



Source: Prashanth Menon

ROF EVALUATION – TPC-H Q19



Source: Prashanth Menon

PARTING THOUGHTS

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

ROF combines vectorization and compilation into a hybrid query processing model.

→ Trades off additional instructions for reduced CPI

NEXT CLASS

Hash Join Implementations