Charting the Design Space of Query Execution using VOILA

Tim Gubner Peter Boncz

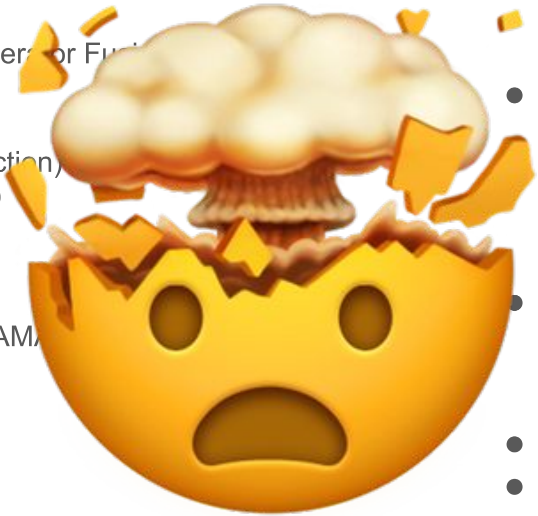
CWI
Database Architectures

Let's Design a HTAP/OLAP System

- Execution paradigm
    - Data-centric
    - Vectorized
    - Mixes (e.g. Relaxed Operator Fusion)
- Selective processing
    - None
    - Selection vector (indirection)
    - Bitmask (SIMD friendly)
    - Mixes
- Prefetching
    - Naive
    - State-machine-based (AMAC, IMV)
- Buffering
    - None
    - Intra-operator
    - Inter-operator
- Adaptivity
    - None
    - Micro (operation level)
    - Macro (operator/plan level)
- Memory layout
    - Columnar
    - Row-wise
    - Mixes (PAX)
- Granularity
    - Column
    - Vector
    - Block
    - Value
    - Partial value
- Compression
    - None
    - "Compressed Execution"
    - Storage
- Different algorithms
- `NULL` handling
- Overflow handling
- ...

Let’s Design a HTAP/OLAP System

- Execution paradigm
  - Data-centric
  - Vectorized
  - Mixes (e.g. Relaxed Oper or Fu
- Selective processing
  - None
  - Selection vector (indirection)
  - Bitmask (SIMD friendly)
  - Mixes
- Prefetching
  - Naive
  - State-machine-based (AM
- Buffering
  - None
  - Intra-operator
  - Inter-operator
- Adaptivity
  - None
  - Micro (operation level)
  - Macro (operator/plan level)



- Memory layout
  - Columnar
  - Row-wise
  - Mixes (PAX)
- Granularity
  - Column
  - Vector
  - Block
  - Value
  - Partial value
- Compression
  - None
  - “Compressed Execution”
  - Storage
- Different algorithms
- `NULL` handling
- Overflow handling
- ...

A Glimpse into our Knowledge

Data-centric (compiled tuple-at-a-time, e.g. Hyper)

- \+ Latency (of single tuple)
- \+ Computation
- \- Compilation time

Vectorized (column-slice-at-a-time, e.g. Vectorwise)

- \+ Parallel memory access
- \+ Adaptivity
- \+ Profiling

A Glimpse into our Knowledge

Data-centric (compiled tuple-at-a-time, e.g. Hyper)

- \+ Latency (of single tuple)
- \+ Computation
- \- Compilation time

Vectorized (column-slice-at-a-time, e.g. Vectorwise)

- \+ Parallel memory access
- \+ Adaptivity
- \+ Profiling

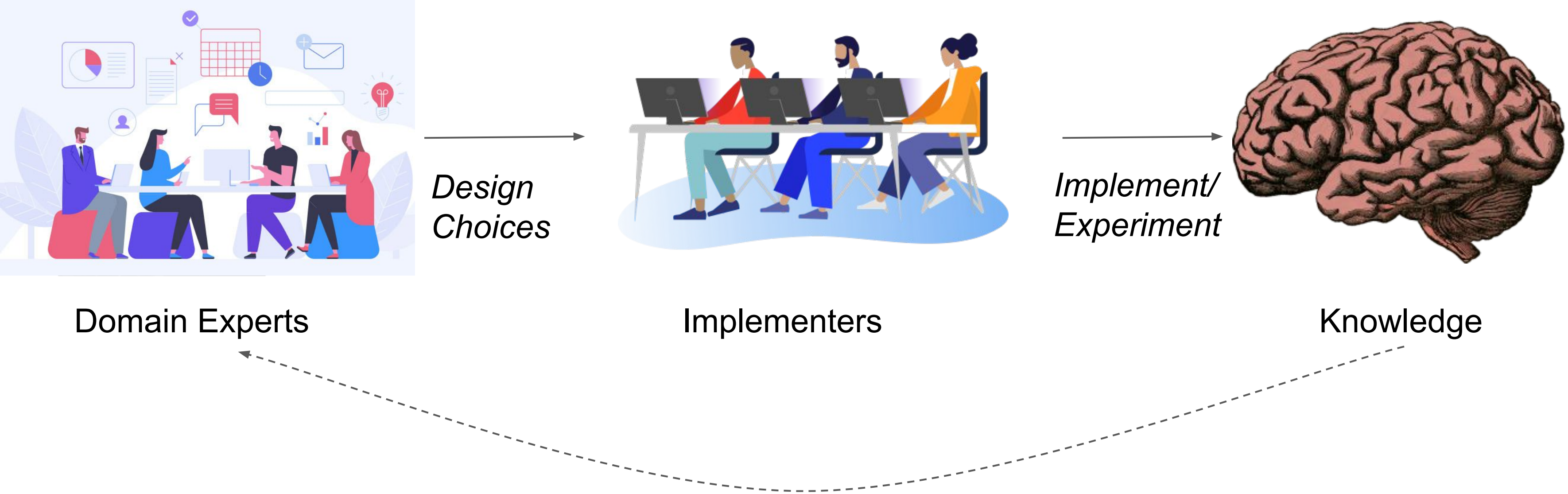
Interaction with Features/Techniques?

- Prefetching always good?
- SIMD always good?
- Hybrids?
- Memory layout?
- Selective processing?
- ...

Interaction with Hardware?

- Huge L3 (> 100MB)?
- Slower cores (< 2 GHz)?
- ARM? RISC-V?
- “Crazy” design decisions (e.g. no L3)?
- Accelerators?
- ...

The State-of-the-Art Discovery Process



Domain Experts

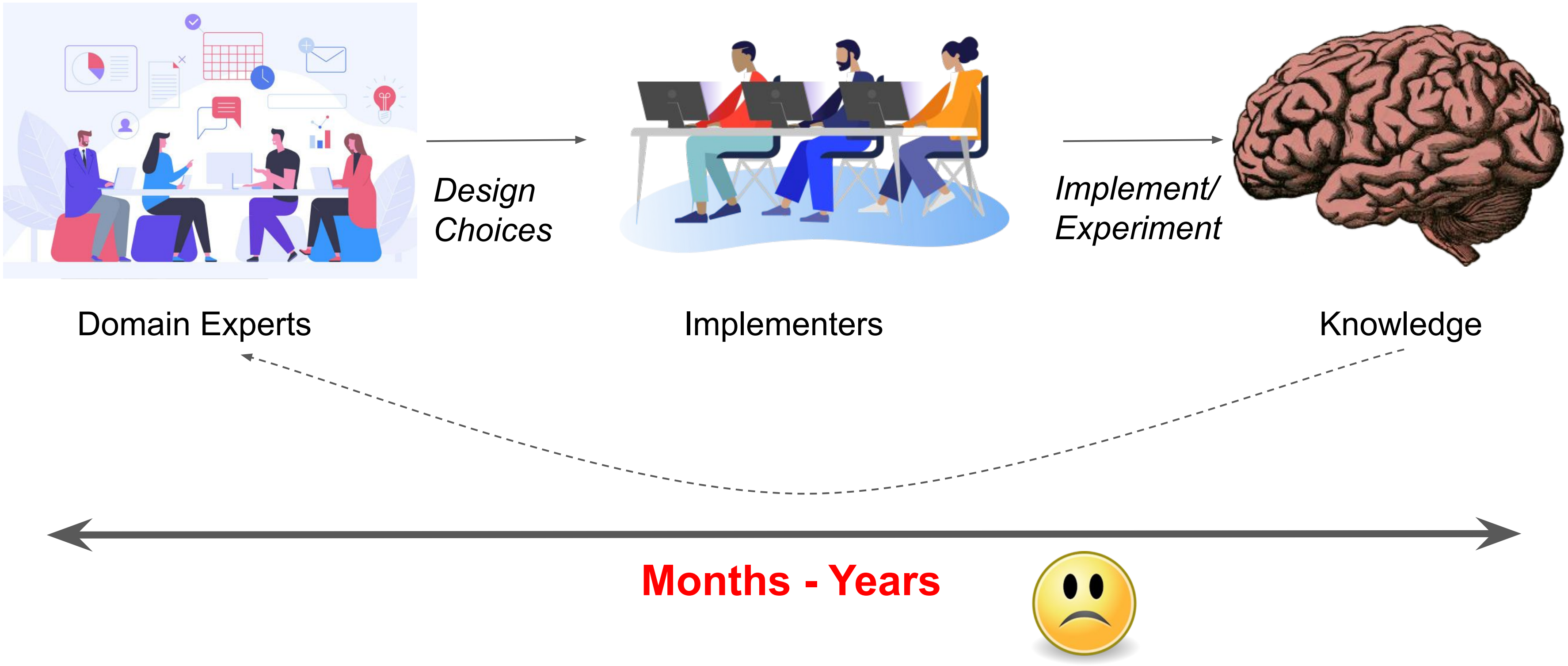
Design Choices

Implementers

Implement/ Experiment

Knowledge

The State-of-the-Art Discovery Process



Design Choices

Implement/ Experiment

Domain Experts

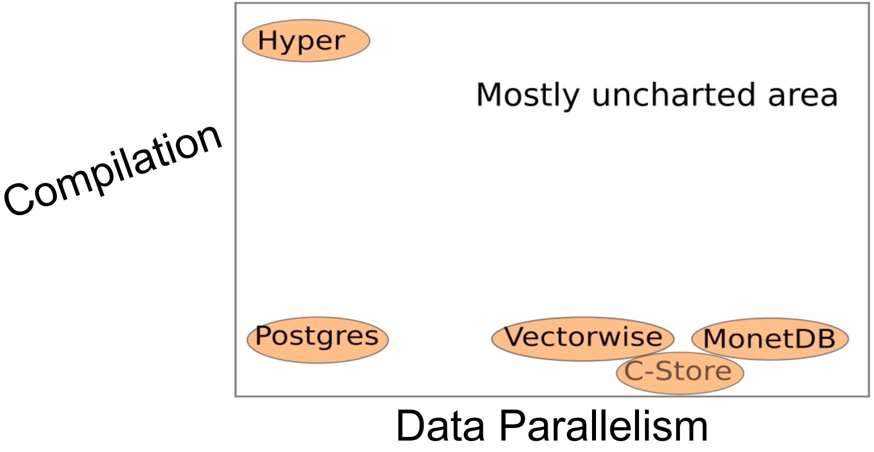
Implementers

Knowledge

Months - Years

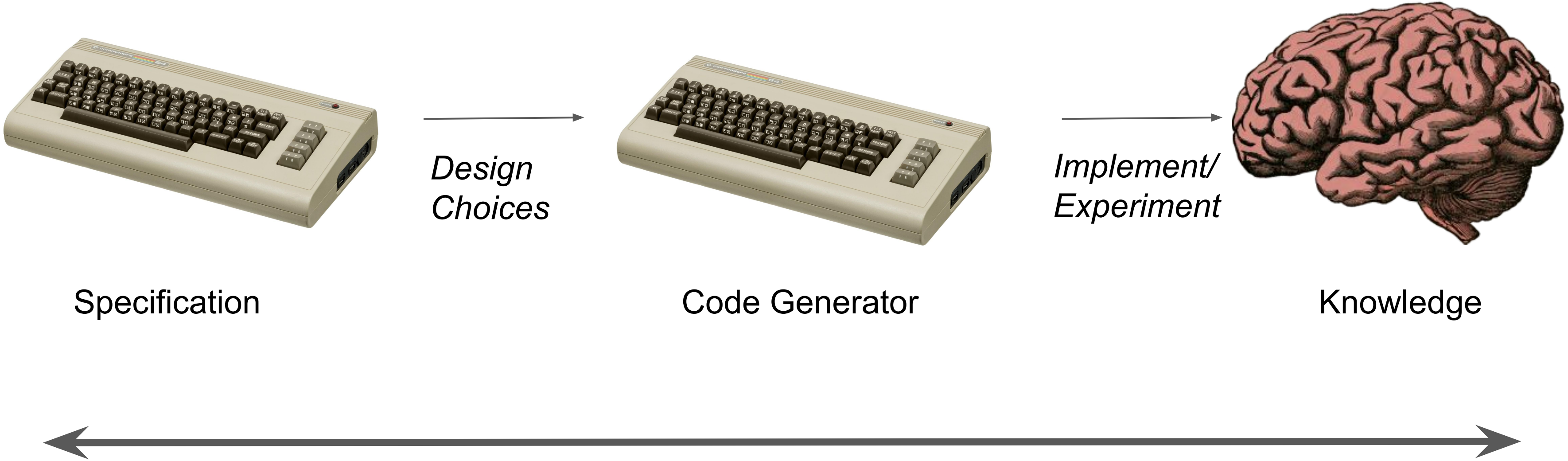
Seeking Diamonds in the Design Space

- Bad reward/risk trade-off
  - High initial investment
  - Low success rate
  - Vast high dimensional space
  - *Some* good points already discovered
- Consequences:
  - Underexplored
  - Understanding = Rules of Thumb
  - Vicious cycle of small improvements



Time for a Change!

The Rise of the Machines



Design Choices

Implement/ Experiment

Specification

Code Generator

Knowledge

Seconds - Minutes

(500,000x - 2,500,000x faster)

Challenges

How can we factor specific details out?

How can we synthesize them, later?

⇨Paper*

*Gubner, Boncz. Charting the Design Space of Query Execution using VOILA. VLDB 2021

Case Study: Hash Join Probe

- Plan: `HashJoin`
- MAL: `j1 := algebra.join(a, b)`
- Low-level plan operators (LOLEPOPs):
  - `FindMatches`
  - `GatherPayload`
- Monad/Monoid comprehensions (e.g. Weld):
  - Dictionary lookup
  - Extract payload
  - Materialize matches

Case Study: Hash Join Probe

- Plan: `HashJoin`
- MAL: `j1 := algebra.join(a, b)`
- Low-level plan operators (LOLEPOPs):
  - `FindMatches`
  - `GatherPayload`
- Monad/Monoid comprehensions (e.g. Weld):
  - Dictionary lookup
  - Extract payload
  - Materialize matches

Too high-level:

- Re-implement high-level operations
- Deforestation problem

Case Study: Hash Join Probe

- Plan: `HashJoin`
- MAL: `j1 := algebra.join(a, b)`
- Low-level plan operators (LOLEPOPs):
    - `FindMatches`
    - `GatherPayload`
- Monad/Monoid comprehensions (e.g. Weld):
    - Dictionary lookup
    - Extract payload
    - Materialize matches

- Imperative:

```
bucket = hash(key) & mask
match = HT.buckets[bucket]
while (match != 0) {
     tmp = HT.key[match]
     if (tmp == key) {
          val = HT.val[match]
          // output (key, val)
     }
     match = HT.next[match]
}
```

Too high-level:

- Re-implement high-level operations
- Deforestation problem

Case Study: Hash Join Probe

- Plan: `HashJoin`
- MAL: `j1 := algebra.join(a, b)`
- Low-level plan operators (LOLEPOPs):
  - `FindMatches`
  - `GatherPayload`
- Monad/Monoid comprehensions (e.g. Weld):
  - Dictionary lookup
  - Extract payload
  - Materialize matches

- Imperative:

```
bucket = hash(key) & mask
match = HT.buckets[bucket]
while (match != 0) {
     tmp = HT.key[match]
     if (tmp == key) {
          val = HT.val[match]
          // output (key, val)
     }
     match = HT.next[match]
}
```

Too high-level:

- Re-implement high-level operations
- Deforestation problem

Too low-level:

- Stuck with one implementation
- Data parallelism?

VOILA

= Variable Operator Implementation LAnguage

Idea:

- Performance-focussed, not necessarily elegant
- Data-parallelism via algorithmic patterns
- Keep operator context
- Low-level, more high-level than C

VOILA

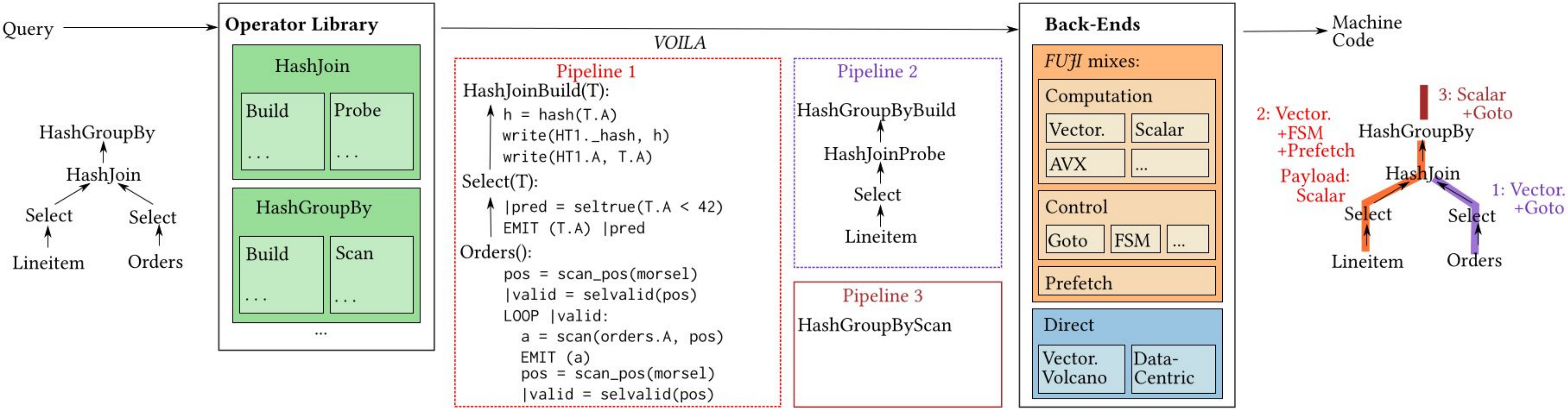
Describes operator implementation

```
hashjoin_probe(child):
  key = child[0]
  hash = hash(key)
  bucket = bucket_lookup(HT, hash)
  hit = seltrue(ne(bucket, 0))
  LOOP | hit:
    k = gather(HT.key, bucket)
    found = seltrue(eq(k, key))
    v = gather(HT.value, bucket) | found
    EMIT (k, v) | found
    bucket = gather(HT.next, bucket)
    hit = seltrue(ne(bucket, 0))
```

Features:

- Predication instead of branching (e.g. `seltrue` creates predicate)
- `LOOP`s
- Specialized statements to move data (`EMIT`)
- Tuples (`[]` and `()`)

VOILA-based Synthesis



Query

HashGroupBy
HashJoin
Select
Select
Lineitem
Orders

Operator Library

HashJoin
Build ...
Probe ...

HashGroupBy
Build ...
Scan ...

...

VOILA

Pipeline 1

```
HashJoinBuild(T):
    h = hash(T.A)
    write(HT1._hash, h)
    write(HT1.A, T.A)
Select(T):
    |pred = seltrue(T.A < 42)
    EMIT (T.A) |pred
Orders():
    pos = scan_pos(morsel)
    |valid = selvalid(pos)
    LOOP |valid:
      a = scan(orders.A, pos)
      EMIT (a)
      pos = scan_pos(morsel)
      |valid = selvalid(pos)
```

Pipeline 2

HashGroupByBuild
HashJoinProbe
Select
Lineitem

Pipeline 3

HashGroupByScan

Back-Ends

FUJI mixes:

Computation
Vector.
Scalar
AVX
...

Control
Goto
FSM
...

Prefetch

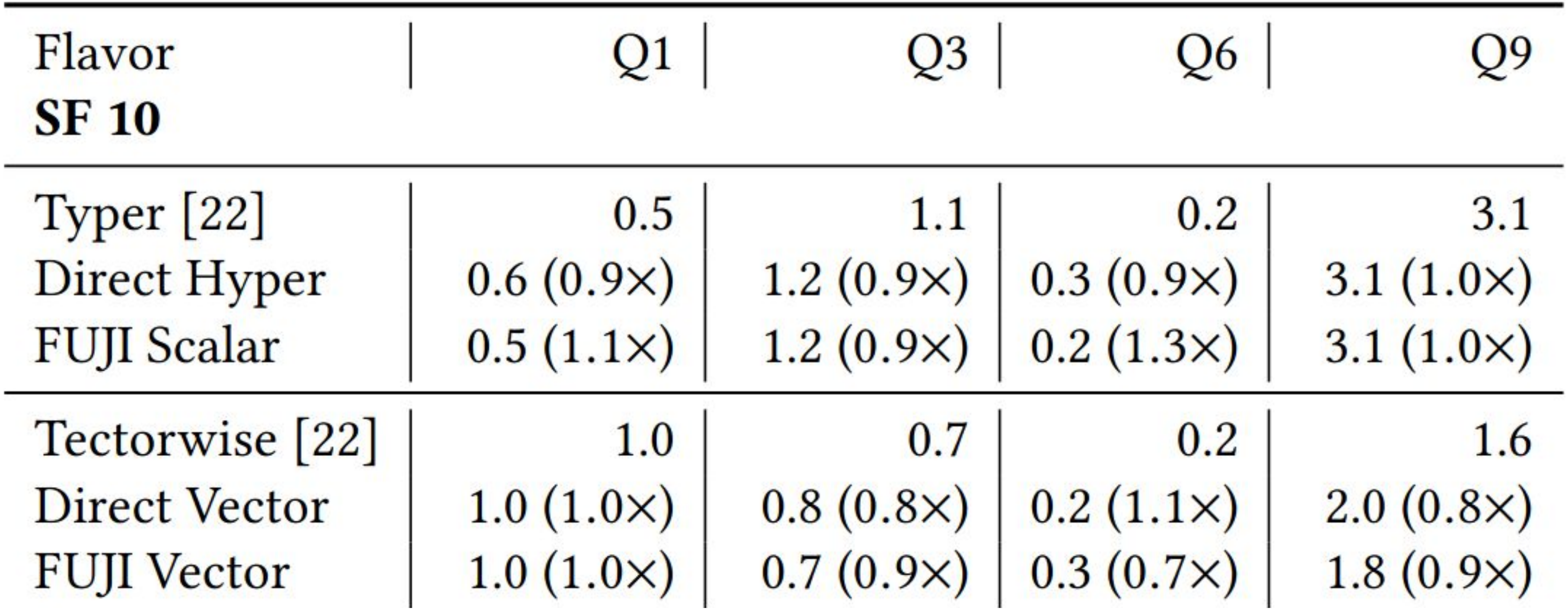
Direct
Vector. Volcano
Data-Centric

Machine Code

3: Scalar +Goto
2: Vector. +FSM +Prefetch
Payload: Scalar
1: Vector. +Goto

HashGroupBy
HashJoin
Select
Select
Lineitem
Orders

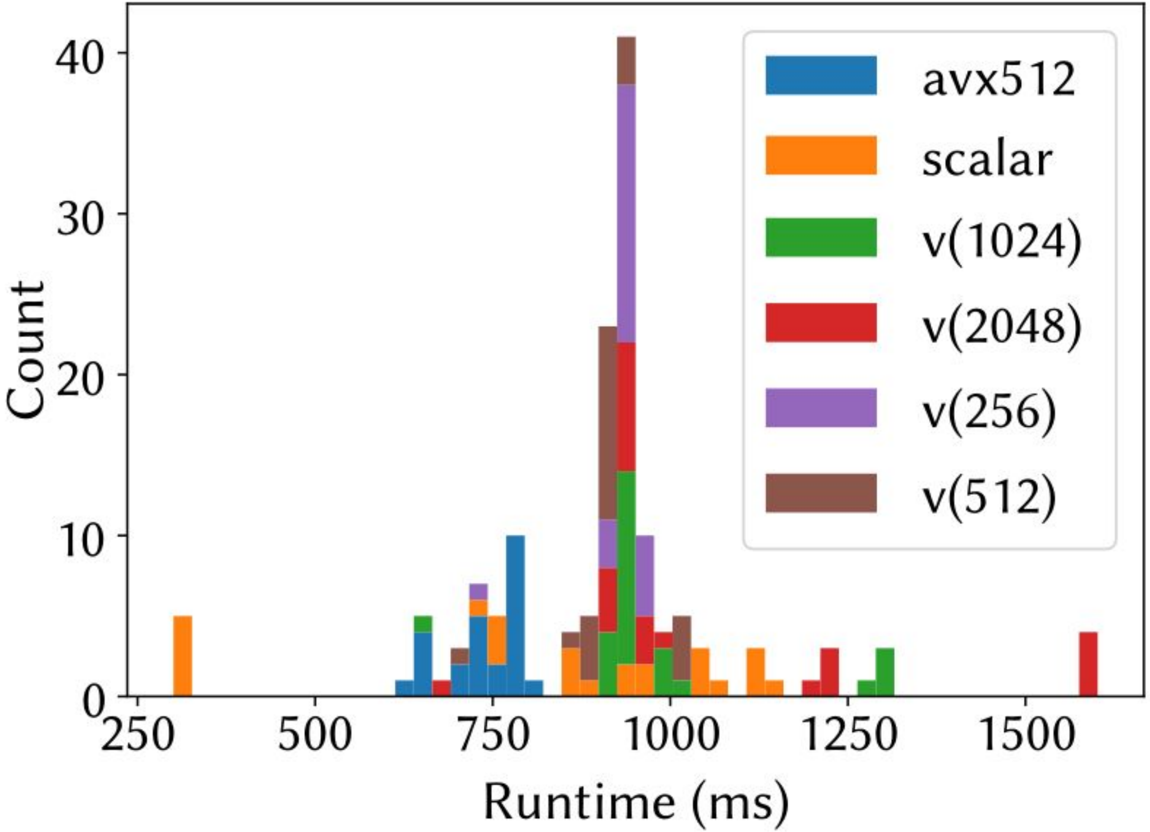
VOILA-gen. Code performs on par with State-of-the-Art



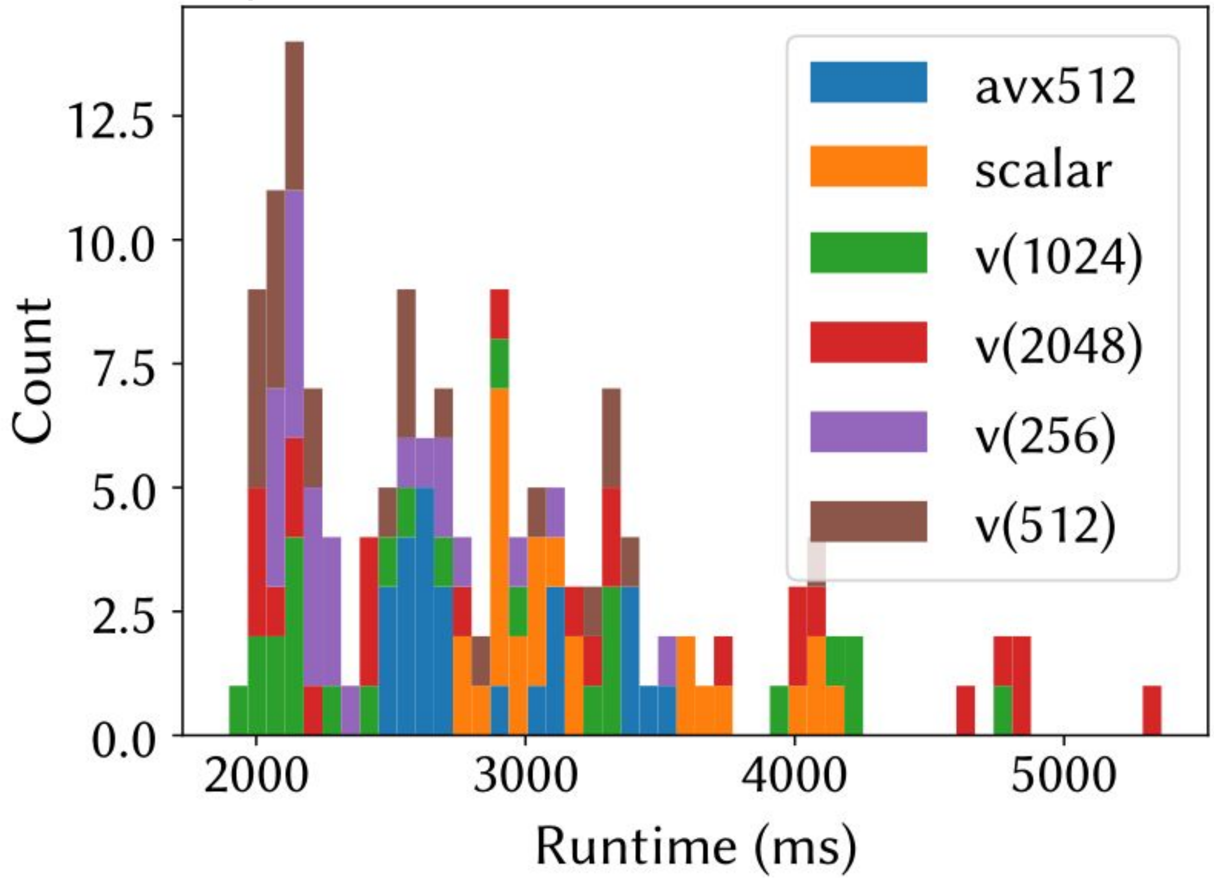
Flavor **SF 10**	Q1	Q3	Q6	Q9
Typer [22]	0.5	1.1	0.2	3.1
Direct Hyper	0.6 (0.9×)	1.2 (0.9×)	0.3 (0.9×)	3.1 (1.0×)
FUJI Scalar	0.5 (1.1×)	1.2 (0.9×)	0.2 (1.3×)	3.1 (1.0×)
Tectorwise [22]	1.0	0.7	0.2	1.6
Direct Vector	1.0 (1.0×)	0.8 (0.8×)	0.2 (1.1×)	2.0 (0.8×)
FUJI Vector	1.0 (1.0×)	0.7 (0.9×)	0.3 (0.7×)	1.8 (0.9×)

[22] T. Kersten et al. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. VLDB 2018

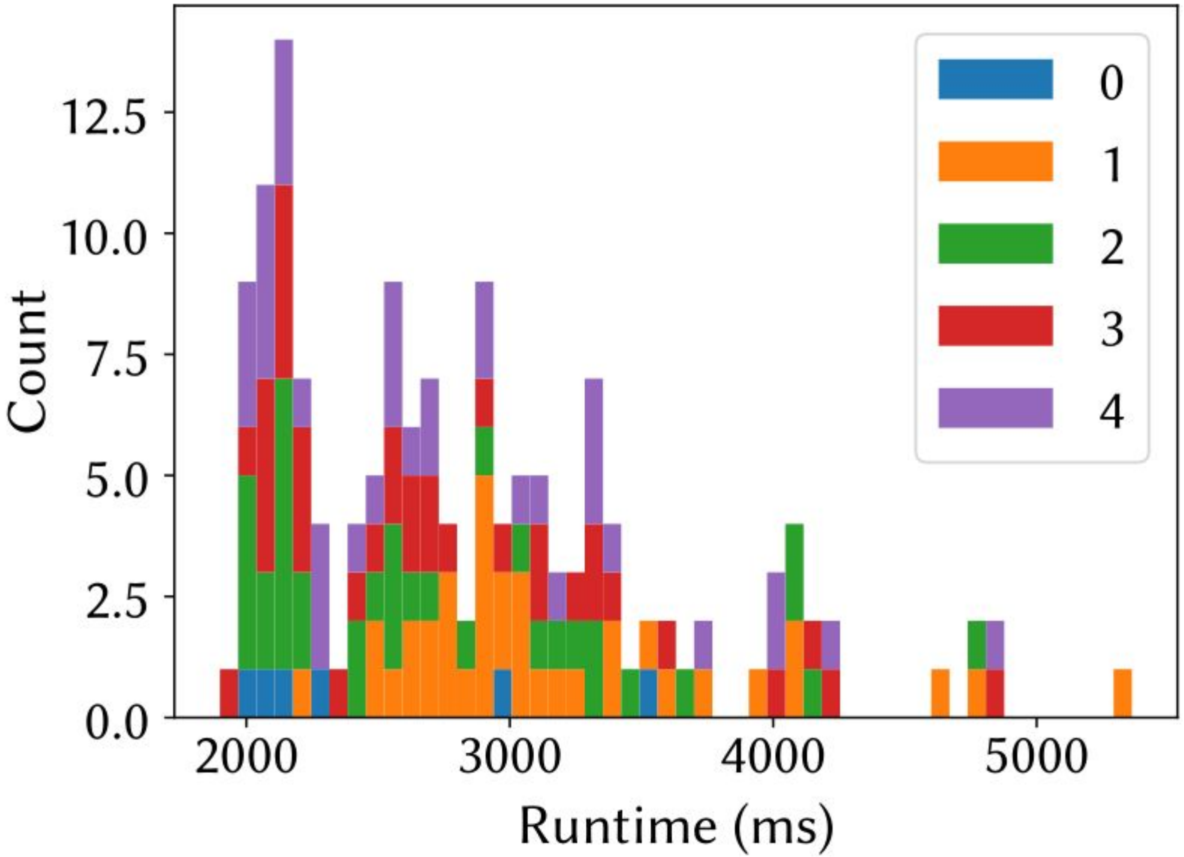
Q1 (Computation)



Q9 (Computation)



Q9 (Prefetch)



Takeaways

With VOILA, we can:

- Encode commonly used operators
- Synthesize **many** different flavors ⇨ semi-automatic exploration
- Get top-notch performance

Future Work:

- More elegant VOILA?
- WCOJs in VOILA?
- More exploration?
- VOILA in practice (Adaptive VM)