Charting the Design Space of Query Execution using VOILA

Tim Gubner

Peter Boncz
Let’s Design a HTAP/OLAP System

- **Execution paradigm**
  - Data-centric
  - Vectorized
  - Mixes (e.g. Relaxed Operator Fusion)
- **Selective processing**
  - None
  - Selection vector (indirection)
  - Bitmap (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 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
- ...
A Glimpse into our Knowledge

<table>
<thead>
<tr>
<th>Data-centric (compiled tuple-at-a-time, e.g. Hyper)</th>
<th>Vectorized (column-slice-at-a-time, e.g. Vectorwise)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Latency (of single tuple)</td>
<td>+ Parallel memory access</td>
</tr>
<tr>
<td>+ Computation</td>
<td>+ Adaptivity</td>
</tr>
<tr>
<td>- Compilation time</td>
<td>+ Profiling</td>
</tr>
</tbody>
</table>
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

Domain Experts

Design Choices

Implementers

Implement/Experiment

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

Specification -> Design Choices -> Code Generator -> Implement/Experiment -> Knowledge

Seconds - Minutes
(500,000x - 2,500,000x faster)
Challenges

How can we factor specific details out?

How can we synthesize them, later?

*Gubner, Boncz. Charting the Design Space of Query Execution using VOILA. VLDB 2021
Case Study: Hash Join Probe

- **Plan:** HashJoin
- **MAL:** $j_1 := \text{algebra.join}(a, b)$
- **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:** \( j1 := \text{algebra.join}(a, b) \)
- **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:**
  
  ```java
  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:** \( j_1 := \text{algebra.join}(a, b) \)
- **Low-level plan operators (LOLEPOPs):**
  - FindMatches
  - GatherPayload
- **Monad/Monoid comprehensions (e.g. Weld):**
  - Dictionary lookup
  - Extract payload
  - Materialize matches

**Imperative:**

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

**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

Features:

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

```python
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))
```
VOILA-based Synthesis
VOILA-gen. Code performs on par with State-of-the-Art

<table>
<thead>
<tr>
<th>Flavor</th>
<th>Q1</th>
<th>Q3</th>
<th>Q6</th>
<th>Q9</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Typer [22]</td>
<td>0.5</td>
<td>1.1</td>
<td>0.2</td>
<td>3.1</td>
</tr>
<tr>
<td>Direct Hyper</td>
<td>0.6 (0.9×)</td>
<td>1.2 (0.9×)</td>
<td>0.3 (0.9×)</td>
<td>3.1 (1.0×)</td>
</tr>
<tr>
<td>FUJI Scalar</td>
<td>0.5 (1.1×)</td>
<td>1.2 (0.9×)</td>
<td>0.2 (1.3×)</td>
<td>3.1 (1.0×)</td>
</tr>
<tr>
<td>Tectorwise [22]</td>
<td>1.0</td>
<td>0.7</td>
<td>0.2</td>
<td>1.6</td>
</tr>
<tr>
<td>Direct Vector</td>
<td>1.0 (1.0×)</td>
<td>0.8 (0.8×)</td>
<td>0.2 (1.1×)</td>
<td>2.0 (0.8×)</td>
</tr>
<tr>
<td>FUJI Vector</td>
<td>1.0 (1.0×)</td>
<td>0.7 (0.9×)</td>
<td>0.3 (0.7×)</td>
<td>1.8 (0.9×)</td>
</tr>
</tbody>
</table>

[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 $\Rightarrow$ semi-automatic exploration
- Get top-notch performance

Future Work:

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