Vectorization vs. Compilation
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
SELECT * FROM foo
WHERE str_col = 'abc'
    AND int_col = 4;

\[ \sigma_{\text{str} \_\text{col} = 'abc' \ \&\& \ \text{int} \_\text{col} = 4} \]

```
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).
**HYPER – JIT QUERY COMPILATION**

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

```c
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)
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)
SINGLE-THREADED PERFORMANCE

Source: Timo Kersten
## Single-threaded Performance

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

- **Auto**
- **Manual**
- **Auto+Manual**

### Reduction of Instr. (%)

- **Q1**: Auto - 29.0, Manual - 12.0, Auto+Manual - 35.0
- **Q9**: Auto - 62.5, Manual - 82.6, Auto+Manual - 82.9
- **Q18**: Auto - 42.0, Manual - 60.1, Auto+Manual - 61.2

Source: Timo Kersten

15-721 (Spring 2020)
AUTO-VECTORIZATION

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

Source: Timo Kersten
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**

![Diagram showing the design space between vectorization and compilation]

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

*Source: Timo Kersten*
Each pipeline **fuses** operators together into loop
Each pipeline is a **tuple-at-a-time** process
Each pipeline **fuses** operators together into loop
Each pipeline is a **tuple-at-a-time** process

```python
def plan(state):
    agg = dict()
    for t in A:
        if t.age > 20:
            agg[t.city]["count"]++
    for t in agg:
        emit(t)
```
Each pipeline **fuses** operators together into loop
Each pipeline is a **tuple-at-a-time** process

```python
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.

```python
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.

```
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.

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

**Cannot SIMD**

**Cannot Prefetch**
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.
ROF EXAMPLE

Scan

Filter

Agg

Emit
ROF EXAMPLE

Vectorization Candidate

Scan

Filter

Agg

Emit

Scan

Filter

Agg

Emit
ROF EXAMPLE

Vectorization Candidate

Stage #2
Stage Buffer
Stage #1
ROF EXAMPLE

```python
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)
```
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

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

- LLVM
- LLVM + ROF

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q3</th>
<th>Q13</th>
<th>Q14</th>
<th>Q19</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM</td>
<td>901</td>
<td>1396</td>
<td>2641</td>
<td>383</td>
<td>540</td>
</tr>
<tr>
<td>LLVM + ROF</td>
<td>892</td>
<td>846</td>
<td>1763</td>
<td>191</td>
<td>220</td>
</tr>
</tbody>
</table>

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

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

Execution Time (ms)

- Interpreted: 21,475 ms
- Compiled: 568 ms
- ROF + SIMD: 196 ms
- ROF + SIMD + Pretching: 189 ms

↓97%  ↓65%  ↓3.5%

Source: Prashanth Menon
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