Query Compilation & Code Generation
LAST CLASS

How to use SIMD to vectorize core database algorithms for sequential scans.
→ Intra-query parallelism

The research literature in the 2010s can give the impression that vectorization and JIT compilation are mutually exclusive.
OPTIMIZATION GOALS

Approach #1: Reduce Instruction Count
→ Use fewer instructions to do the same amount of work.

Approach #2: Reduce Cycles per Instruction
→ Execute more CPU instructions in fewer cycles.

Approach #3: Parallelize Execution
→ Use multiple threads to compute each query in parallel.
After minimizing the disk I/O during query execution, the only way to increase throughput is to reduce the number of instructions executed.

→ To go \textbf{10x} faster, the DBMS must execute 90\% fewer instructions.
→ To go \textbf{100x} faster, the DBMS must execute 99\% fewer instructions.
TODAY’S AGENDA

Background
Source-to-Source Compilation / Transpilation
JIT Compilation
Real-world Implementations
Project Status Discussion
One way to achieve a significant reduction in instructions is through **code specialization**.

This means generating code that is specific to a task in the DBMS (e.g., one query).

Most code is written to make it easy for humans to understand rather than performance...
**QUERY INTERPRETATION**

```
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
 FROM B
 WHERE B.val = ? + 1
 GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
```
SELECT *  
FROM A, C,  
(SELECT B.id, COUNT(*)  
FROM B  
WHERE B.val = ? + 1  
GROUP BY B.id) AS B  
WHERE A.val = 123  
AND A.id = C.a_id  
AND B.id = C.b_id
**EXPRESSION EVALUATION**

**Execution Context**

- **Current Tuple**: (123, 1000)
- **Query Parameters**: (int:999)
- **Table Schema**: B→(int:id, int:val)

**SELECT** *
FROM A, C,
(SELECT B.id, COUNT(*)
FROM B
WHERE B.val = ? + 1
GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
EXECUTION CONTEXT

```
SELECT * 
FROM A, C, 
(SELECT B.id, COUNT(*) 
FROM B 
WHERE B.val = ? + 1 
GROUP BY B.id) AS B 
WHERE A.val = 123 
AND A.id = C.a_id 
AND B.id = C.b_id
```
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
FROM B
WHERE B.val = ? + 1
GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id

EXPRESSION EVALUATION

Execution Context

Current Tuple (123, 1000)
Query Parameters (int:999)
Table Schema B→(int:id, int:val)

Attribute(S.val)
Op(=)
1000
Paramter($1)
Op(+)  Constant(1)
SELECT * 
FROM A, C,  
(SELECT B.id, COUNT(*) 
FROM B 
WHERE B.val = ? + 1 
GROUP BY B.id) AS B 
WHERE A.val = 123 
AND A.id = C.a_id 
AND B.id = C.b_id
```
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
FROM B
WHERE B.val = ? + 1
GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
```
SELECT *
FROM A, C,
 (SELECT B.id, COUNT(*)
  FROM B
  WHERE B.val = ? + 1
  GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
**EXPRESSION EVALUATION**

**Execution Context**

- **Current Tuple**: (123, 1000)
- **Query Parameters**: (int:999)
- **Table Schema**: B→(int:id, int:val)

```
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
 FROM B
 WHERE B.val = ? + 1
 GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
```
The DBMS generates code for any CPU-intensive task that has a similar execution pattern on different inputs.

→ Access Methods
→ Stored Procedures
→ Query Operator Execution
→ Predicate Evaluation  Most Common
→ Logging Operations

For query-focused compilation, the DBMS (typically) specializes it after generating the physical plan for a query.
CODE SPECIALIZATION BENEFITS

Attribute types are known *a priori*.
→ Data access function calls can be converted to inline pointer casting.

Predicates are known *a priori*.
→ They can be evaluated using primitive data comparisons.

No function calls in loops
→ Allows the compiler to efficiently distribute data to registers and increase cache reuse.
CODE SPECIALIZATION METHODS

Approach #1: Transpilation
→ Write code that converts a relational query plan into imperative language *source code* and then run it through a conventional compiler to generate native code.

Approach #2: JIT Compilation
→ Generate an *intermediate representation* (IR) of the query that the DBMS then compiles into native code.
HIQUE: HOLISTIC CODE GENERATION

For a given query plan, create a C/C++ program that implements that query’s execution.
→ Bake in all the predicates and type conversions.

Use an off-shelf compiler to convert the code into a shared object, link it to the DBMS process, and then invoke the exec function.
HIQUE: OPERATOR TEMPLATES

Interpreted Plan

```python
for t in range(table.num_tuples):
    tuple = get_tuple(table, t)
    if eval(predicate, tuple, params):
        emit(tuple)
```

1. Get schema in catalog for table.
2. Calculate offset based on tuple size.
3. Return pointer to tuple.
HIQUE: OPERATOR TEMPLATES

**Interpreted Plan**

```python
for t in range(table.num_tuples):
    tuple = get_tuple(table, t)
    if eval(predicate, tuple, params):
        emit(tuple)
```

1. Get schema in catalog for table.
2. Calculate offset based on tuple size.
3. Return pointer to tuple.

1. Traverse predicate tree and pull values up.
2. If tuple value, calculate the offset of the target attribute.
3. Perform casting as needed for comparison operators.
4. Return true / false.
HIQUE: OPERATOR TEMPLATES

Interpreted Plan

```python
for t in range(table.num_tuples):
    tuple = get_tuple(table, t)
    if eval(predicate, tuple, params):
        emit(tuple)
```

1. Get schema in catalog for table.
2. Calculate offset based on tuple size.
3. Return pointer to tuple.

Templated Plan

```python
tuple_size = ###
predicate_offset = ###
parameter_value = ###
```

```python
for t in range(table.num_tuples):
    tuple = table.data + t * tuple_size
    val = (tuple+predicate_offset)
    if (val == parameter_value + 1):
        emit(tuple)
```

1. Traverse predicate tree and pull values up.
2. If tuple value, calculate the offset of the target attribute.
3. Perform casting as needed for comparison operators.
4. Return true / false.
HIQUE: OPERATOR TEMPLATES

Interpreted Plan

for t in range(table.num_tuples):
    tuple = get_tuple(table, t)
    if eval(predicate, tuple, params):
        emit(tuple)

1. Get schema in catalog for table.
2. Calculate offset based on tuple size.
3. Return pointer to tuple.

Templated Plan

tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
    tuple = table.data + t * tuple_size
    val = (tuple + predicate_offset)
    if (val == parameter_value + 1):
        emit(tuple)

1. Traverse predicate tree and pull values up.
2. If tuple value, calculate the offset of the target attribute.
3. Perform casting as needed for comparison operators.
4. Return true / false.
HIQUE: DBMS INTEGRATION

The generated query code can invoke any other function in the DBMS. This allows it to use all the same components as interpreted queries.

→ Network Handlers
→ Buffer Pool Manager
→ Concurrency Control
→ Logging / Checkpoints
→ Indexes

Debugging is (relatively) easy because you step through the generated source code.
HIQUE: EVALUATION

Generic Iterators
→ Canonical model with generic predicate evaluation.

Optimized Iterators
→ Type-specific iterators with inline predicates.

Generic Hardcoded
→ Handwritten code with generic iterators/predicates.

Optimized Hardcoded
→ Direct tuple access with pointer arithmetic.

HIQUE
→ Query-specific specialized code.
QUERY COMPILATION EVALUATION

Intel Core 2 Duo 6300 @ 1.86GHz
Join Query: 10k ⨝ 10k → 10m

Source: Konstantinos Krikellas
QUERY COMPILATION COST

Intel Core 2 Duo 6300 @ 1.86GHz
TPC-H Queries (Scalefactor=1)

- Compile (-O0)
- Compile (-O2)

Compilation Time (ms)

Source: Konstantinos Krikellas
Relational operators are a useful way to reason about a query but are not the most efficient way to execute it.

It takes a (relatively) long time to compile a C/C++ source file into executable code.

HIQUE also does not support for full pipelining.
HYPER: JIT QUERY COMPILATION

Compile queries in-memory into native code using the LLVM toolkit.
→ Instead of emitting C++ code, HyPer emits LLVM IR.

Aggressive operator fusion within pipelines to keep a tuple in CPU registers for as long as possible.
→ Push-based vs. Pull-based
→ Data Centric vs. Operator Centric
**PIPELINED OPERATORS**

```sql
SELECT * 
FROM A, C, 
     (SELECT B.id, COUNT(*) 
      FROM B 
      WHERE B.val = ? + 1 
      GROUP BY B.id) AS B 
WHERE A.val = 123 
AND A.id = C.a_id 
AND B.id = C.b_id
```

**Pipeline Boundaries**

- **#1**: 
  - A
  - A.id = C.a_id
  - A.val = 123

- **#2**: 
  - B
  - B.val = ? + 1

- **#3**: 
  - B.id, COUNT(*)

- **#4**: 
  - C
  - B.id = C.b_id
**PUSH-BASED EXECUTION**

**Generated Query Plan**

```sql
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
FROM B
WHERE B.val = ? + 1
GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
```

- **for t in A:**
  - if t.val == 123:
    - Materialize t in HashTable \(\bowtie(A.id=C.a_id)\)
- **for t in B:**
  - if t.val == <param> + 1:
    - Aggregate t in HashTable \(\Gamma(B.id)\)
- **for t in \(\Gamma(B.id)\):**
  - Materialize t in HashTable \(\bowtie(B.id=C.b_id)\)
- **for t3 in C:**
  - for t2 in \(\bowtie(B.id=C.b_id)\):
    - for t1 in \(\bowtie(A.id=C.a_id)\):
      - emit(t1 \(\bowtie\) t2 \(\bowtie\) t3)
```
PUSH-BASED EXECUTION

Generated Query Plan

for t in A:
  if t.val == 123:
    Materialize t in HashTable \(\Join\)(A.id=C.a_id)

for t in B:
  if t.val == <param> + 1:
    Aggregate t in HashTable \(\Gamma\)(B.id)

for t in \(\Gamma\)(B.id):
  Materialize t in HashTable \(\Join\)(B.id=C.b_id)

for t3 in C:
  for t2 in \(\Join\)(B.id=C.b_id):
    for t1 in \(\Join\)(A.id=C.a_id):
      emit(t1\Join t2\Join t3)
**QUERY COMPILATION EVALUATION**

*Dual Socket Intel Xeon X5770 @ 2.93GHz*

*TPC-H Queries (Scalefactor=1)*

- **Oracle**
- **MonetDB**
- **VectorWise**
- **HyPer (C++)**
- **HyPer (LLVM)**

**Source:** Thomas Neumann

---

**Execution Time (ms)**

<table>
<thead>
<tr>
<th>Query</th>
<th>Oracle</th>
<th>MonetDB</th>
<th>VectorWise</th>
<th>HyPer (C++)</th>
<th>HyPer (LLVM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>4221</td>
<td>72</td>
<td>98</td>
<td>142</td>
<td>35</td>
</tr>
<tr>
<td>Q2</td>
<td>6555</td>
<td>218</td>
<td>374</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>16410</td>
<td>112</td>
<td>257</td>
<td>141</td>
<td>80</td>
</tr>
<tr>
<td>Q4</td>
<td>3830</td>
<td>436</td>
<td>203</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td>15212</td>
<td>1107</td>
<td>1416</td>
<td>1105</td>
<td></td>
</tr>
</tbody>
</table>

*Crashed!*
QUERY COMPILATION COST

HIQUE (-O2) vs. HyPer
TPC-H Queries

<table>
<thead>
<tr>
<th></th>
<th>HIQUE</th>
<th>HyPer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>274</td>
<td>13</td>
</tr>
<tr>
<td>Q2</td>
<td>403</td>
<td>37</td>
</tr>
<tr>
<td>Q3</td>
<td>619</td>
<td>15</td>
</tr>
</tbody>
</table>

Source: Konstantinos Krikellas

Warning: Not a scientific comparison!
OBSERVATION

HyPer's query compilation time grows super-linearly relative to the query size.
- # of joins
- # of predicates
- # of aggregations

Not a big issue with OLTP applications.
Major problem with OLAP workloads.
HYPER: ADAPTIVE EXECUTION

Generate LLVM IR for the query and immediately start executing the IR using an interpreter. Then the DBMS compiles the query in the background.

When the compiled query is ready, seamlessly replace the interpretive execution.

→ For each morsel, check to see whether the compiled version is available.
HYPER: ADAPTIVE EXECUTION

SQL Query

Optimizer (0.2 ms)

Query Plan

Code Generator (0.7 ms)

query Plan

LLVM IR

Byte Code Compiler (0.4 ms)

Byte Code

LLVM IR

Unoptimized LLVM Compiler (6 ms)

x86 Code

LLVM IR

Optimized LLVM Compiler (17 ms)

x86 Code

Source: Andre Kohn
HYPER: ADAPTIVE EXECUTION

AMD Ryzen 7 1700X @ 3.4GHz (One Thread)

TPC-H Queries

| Source: Andre Kohn |

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
</tr>
</thead>
<tbody>
<tr>
<td>858</td>
<td>161</td>
<td>323</td>
<td>352</td>
<td>362</td>
</tr>
<tr>
<td>77</td>
<td>94</td>
<td>104</td>
<td>67</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>80</td>
<td>45</td>
<td>37</td>
</tr>
</tbody>
</table>

Execution Time (ms)

- Byte Code
- Unoptimized LLVM
- Optimized LLVM
# REAL-WORLD IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>Transpilation</th>
<th>Custom</th>
<th>JVM-based</th>
<th>LLVM-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Redshift</td>
<td>IBM System R</td>
<td>Spark</td>
<td>SingleStore</td>
</tr>
<tr>
<td>Oracle</td>
<td>Actian Vector</td>
<td>Neo4j</td>
<td>VitesseDB</td>
</tr>
<tr>
<td></td>
<td>SQLite</td>
<td>Presto / Trino</td>
<td>CMU Peloton</td>
</tr>
<tr>
<td></td>
<td>TUM HyPer</td>
<td>OrientDB</td>
<td>CMU NoisePage</td>
</tr>
<tr>
<td></td>
<td>TUM Umbra</td>
<td>Tajo</td>
<td>TUM LingoDB</td>
</tr>
<tr>
<td></td>
<td>QuestDB</td>
<td>Derby</td>
<td></td>
</tr>
</tbody>
</table>
IBM SYSTEM R

A primitive form of code generation and query compilation was used by IBM in 1970s.
→ Compiled SQL statements into assembly code by selecting code templates for each operator.

Technique was abandoned when IBM built SQL/DS and DB2 in the 1980s:
→ High cost of external function calls
→ Poor portability
→ Software engineer complications
A primitive form of code generation and compilation was used by IBM in 1970s:

→ Compiled SQL statements into assembly code templates for each operator.

Technique was abandoned when IBM built SQL/DS and DB2 in the 1980s:

→ High cost of external function calls
→ Poor portability
→ Software engineer complications
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
**VECTORWISE: PRECOMPILED PRIMITIVES**

```
SELECT * FROM foo
WHERE str_col = 'abc'
  AND int_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);
}
```
Convert query fragments into templated C++ code. → Push-based execution with vectorization.

DBMS checks whether there are already exists a compiled version of each templated fragment in the customer's local cache.

If fragment does not exist in the local cache, then it checks a global cache for the entire fleet of Redshift customers.
ORACLE

Convert PL/SQL stored procedures into Pro*C code and then compiled into native C/C++ code.

They also put Oracle-specific operations directly in the SPARC chips as co-processors.

→ Memory Scans
→ Bit-pattern Dictionary Compression
→ Vectorized instructions designed for DBMSs
→ Security/encryption
MICROSOFT HEKATON

Can compile both procedures and SQL.
→ Non-Hekaton queries can access Hekaton tables through compiled inter-operators.

Generates C code from an imperative syntax tree, compiles it into DLL, and links at runtime.

Employs safety measures to prevent somebody from injecting malicious code in a query.
SQLite

DBMS converts a query plan into opcodes, and then executes them in a custom VM (bytecode engine).
→ Also known as "Virtual DataBase Engine" (VDBE)
→ Opcode specification can change across versions.

SQLite's VM ensures that queries execute the same in any possible environment.

```
sqlite> explain SELECT 1 + 1;
addr opcode p1 p2 p3 p4 p5 comment
---- ----------- ---- ---- ---- ---- --------------
0 Init 0 4 0 0 0 Start at 4
1 Add 2 2 1 0 0 r[1]=r[2]+r[2]
2 ResultRow 1 1 0 0 0 output=r[1]
3 Halt 0 0 0 0 0
4 Integer 1 2 0 0 0 r[2]=1
5 Goto 0 1 0 0 0
Run Time: real 0.000 user 0.000185 sys 0.000000
```
Instead of implementing a separate bytecode interpreter, Umbra's "FlyingStart" adaptive execution framework generates custom IR that maps to x86 assembly in a single pass. → Manually performs dead code elimination. → The DBMS is a basically compiler.

They also wrote their own debugger!
Instead of implementing a separate bytecode interpreter, Umbra's "FlyingStart" adaptive execution framework generates custom IR that maps to x86 assembly in a single pass.

- Manually performs dead code elimination.
- The DBMS is a basically compiler.

They also wrote their own debugger!
There are several JVM-based DBMSs that contain custom code that emits JVM bytecode directly.

→ Spark
→ Neo4j
→ Splice Machine
→ Presto / Trino
→ Derby
→ Tajo

This functionally the same as generating LLVM IR.
APACHE SPARK

Introduced in the new Tungsten engine in 2015. The system converts a query's WHERE clause expression trees into Scala ASTs. It then compiles these ASTs to generate JVM bytecode, which is then executed natively.

Databricks abandoned this approach with their new Photon engine in late 2010s.
**QUESTDB**

Java-based time-series columnar DBMS. The Java front-end converts **WHERE** clause predicates into IR and then uses a C++ backend to compile the IR into vectorized machine code using **asmjit**.

![Execution Time Chart](chart.png)

Source: [Andrey Pechkurov](https://example.com)
SINGLESTORE (PRE-2016)

Performs the same C/C++ code generation as HIQUE and then invokes gcc.

Converts all queries into a parameterized form and caches the compiled query plan.

```
SELECT * FROM A WHERE A.id = 123

SELECT * FROM A WHERE A.id = ?

SELECT * FROM A WHERE A.id = 456
```
A query plan is converted into an imperative plan expressed in a high-level imperative DSL.

→ MemSQL Programming Language (MPL)
→ Think of this as a C++ dialect.

DBMS then converts DSL into custom opcodes.
→ MemSQL Bit Code (MBC)
→ Think of this as JVM byte code.

Lastly, the DBMS compiles the opcodes into LLVM IR and then to native code.

Source: Drew Paroski
**POSTGRESQL**

Added support in 2018 (v11) for JIT compilation of predicates and tuple deserialization with LLVM. → Relies on optimizer estimates to determine when to compile expressions.

Automatically compiles Postgres' back-end C code into LLVM C++ code to remove iterator calls.

Source: Dmitry Melnik
**JITed expressions**

- directly emit LLVM IR for common opcodes
- emit calls to functions implementing less common opcodes
  - can be inlined
- indirect opcode → opcode jumps become direct
- indirect funcexpr calls become direct
  - can be inlined

**TPCH Q01 non-jitted vs jitted:**
- 28759 ms vs 22309 ms
- branch misses: 0.38% vs 0.07%
- iTLB load misses: 58,903,279 vs 48,986 (yes, really)

Source: Dmitry Melnik
VITESSEDB

Query accelerator for Postgres/Greenplum that uses LLVM + intra-query parallelism.
→ JIT predicates
→ Push-based processing model
→ Indirect calls become direct or inlined.
→ Leverages hardware for overflow detection.

Does not support all of Postgres’ types and functionalities. All DML operations are still interpreted.

Source: CK Tan
SingleStore-style conversion of query plans into a database-oriented DSL.

Then compile the DSL into opcodes.

HyPer-style interpretation of opcodes while compilation occurs in the background with LLVM.
fun main() -> int {
  var ret = 0
  for (row in foo) {
    if (row.colA >= 50 and row.colB < 100000) {
      ret = ret + 1
    }
  }
  return ret
}
SELECT * FROM foo
WHERE colA >= 50
AND colB < 100000;

fun main() -> int {
    var ret = 0
    for (row in foo) {
        if (row.colA >= 50 and row.colB < 100000) {
            ret = ret + 1
        }
    }
    return ret
}
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 based on system architecture.

→ Example: Hash join algorithm is the same, but the systems use different hash functions (Murmur2 vs. CRC32×2)
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. Computationally inexpensive.
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

15-721 (Spring 2024)

| Q1   | Fixed-point arithmetic, 4-group aggregation |
| Q6   | Selective filters. Computationally inexpensive. |
| 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

Intel Core i9-7900X (10 cores × 2HT)
TPC-H Queries (Scalefactor=1)

Source: Timo Kersten

<table>
<thead>
<tr>
<th>Query</th>
<th>Tectorwise</th>
<th>Typer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>85</td>
<td>49</td>
</tr>
<tr>
<td>Q6</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Q3</td>
<td>45</td>
<td>48</td>
</tr>
<tr>
<td>Q9</td>
<td>147</td>
<td>111</td>
</tr>
<tr>
<td>Q18</td>
<td>152</td>
<td>99</td>
</tr>
</tbody>
</table>
SINGLE-THREADED PERFORMANCE

Intel Core i9-7900X (10 cores × 2HT)
TPC-H Queries (Scalefactor=1)

- **Tectorwise**
- **Typer**

<table>
<thead>
<tr>
<th>Query</th>
<th>Tectorwise</th>
<th>Typer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>85</td>
<td>49</td>
</tr>
<tr>
<td>Q6</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Q3</td>
<td>45</td>
<td>48</td>
</tr>
<tr>
<td>Q9</td>
<td>111</td>
<td>147</td>
</tr>
<tr>
<td>Q18</td>
<td>152</td>
<td>99</td>
</tr>
</tbody>
</table>

Source: Timo Kersten
## SINGLE-THREADED PERFORMANCE

<table>
<thead>
<tr>
<th></th>
<th>Runtine</th>
<th>Cycles</th>
<th>IPC</th>
<th>Instr.</th>
<th>L1 Miss</th>
<th>LLC Miss</th>
<th>Br. Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>TWise</td>
<td>85</td>
<td>59</td>
<td>2.8</td>
<td>162</td>
<td>2.0</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>Typer</td>
<td>48</td>
<td>34</td>
<td>2.0</td>
<td>68</td>
<td>0.6</td>
<td>0.57</td>
</tr>
<tr>
<td>Q6</td>
<td>TWise</td>
<td>15</td>
<td>11</td>
<td>1.4</td>
<td>15</td>
<td>0.2</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>Typer</td>
<td>16</td>
<td>11</td>
<td>1.8</td>
<td>20</td>
<td>0.3</td>
<td>0.35</td>
</tr>
<tr>
<td>Q3</td>
<td>TWise</td>
<td>45</td>
<td>24</td>
<td>1.8</td>
<td>42</td>
<td>0.9</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>Typer</td>
<td>48</td>
<td>25</td>
<td>0.8</td>
<td>21</td>
<td>0.5</td>
<td>0.16</td>
</tr>
<tr>
<td>Q9</td>
<td>TWise</td>
<td>111</td>
<td>56</td>
<td>1.3</td>
<td>76</td>
<td>2.1</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>Typer</td>
<td>147</td>
<td>74</td>
<td>0.6</td>
<td>42</td>
<td>1.7</td>
<td>0.46</td>
</tr>
<tr>
<td>Q18</td>
<td>TWise</td>
<td>152</td>
<td>48</td>
<td>2.1</td>
<td>102</td>
<td>1.9</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>Typer</td>
<td>99</td>
<td>30</td>
<td>1.6</td>
<td>46</td>
<td>0.8</td>
<td>0.19</td>
</tr>
</tbody>
</table>
Both models are efficient and achieve roughly the same performance. → 100x faster than row-oriented DBMSs!

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

Vectorization is slightly better at hiding cache miss latencies.
PARTING THOUGHTS

Query compilation makes a difference but is non-trivial to implement.

The 2016 version of SingleStore is the best query compilation implementation out there in terms of performance and engineering…
→ Umbra FlyingStart is ridiculously good but that's because the Germans are ridiculously good.

Newer systems choose to implement Vectorwise-style vectorization instead of compilation.
NEXT CLASS

Query Task Scheduling! More Germans!