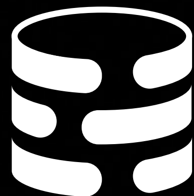
ADVANCED
DATABASE
SYSTEMS



Query Compilation & Code Generation

07

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

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.

MICROSOFT REMARK

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 **10x** faster, the DBMS must execute **90%** fewer instructions.

→ To go **100x** faster, the DBMS must execute **99%** fewer instructions.

COMPILATION IN THE MICROSOFT SQL SERVER HEKATON ENGINE
IEEE DATA ENGINEERING BULLETIN 2011

TODAY'S AGENDA

Background
Source-to-Source Compilation / Transpilation
JIT Compilation
Real-world Implementations
Project Status Discussion

OBSERVATION

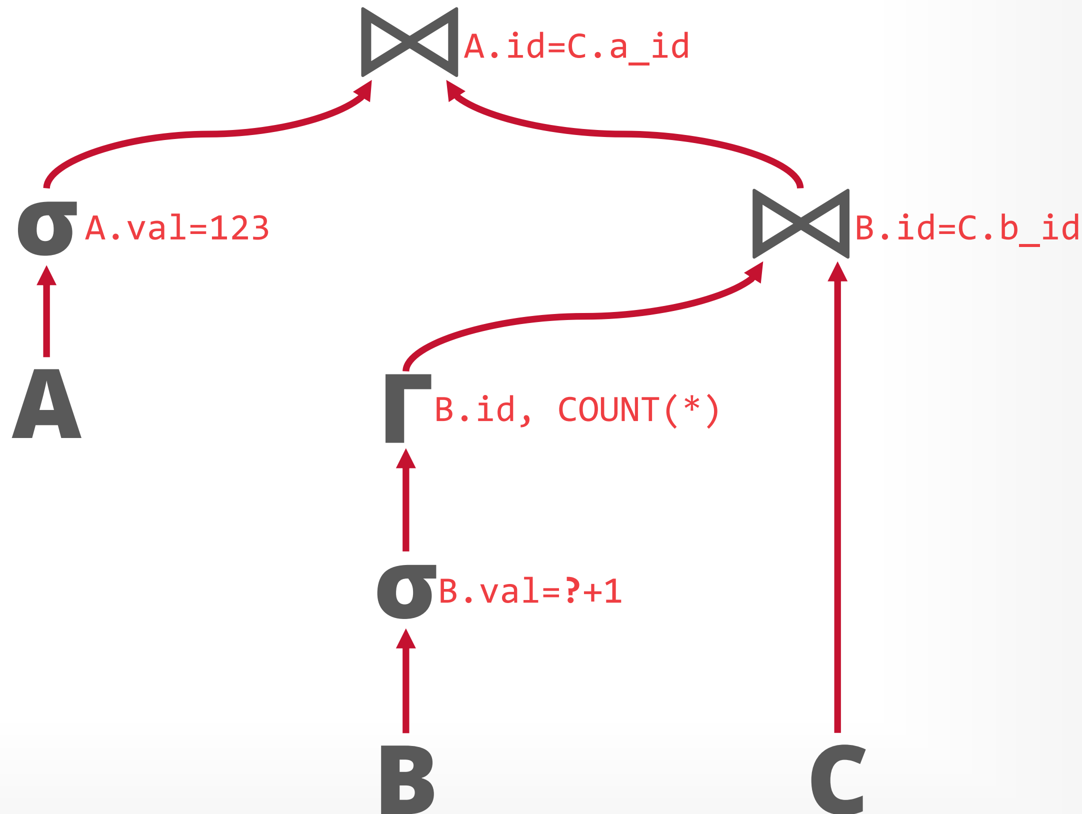
One way to achieve a significant reduction in instructions is through **code specialization**.

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…

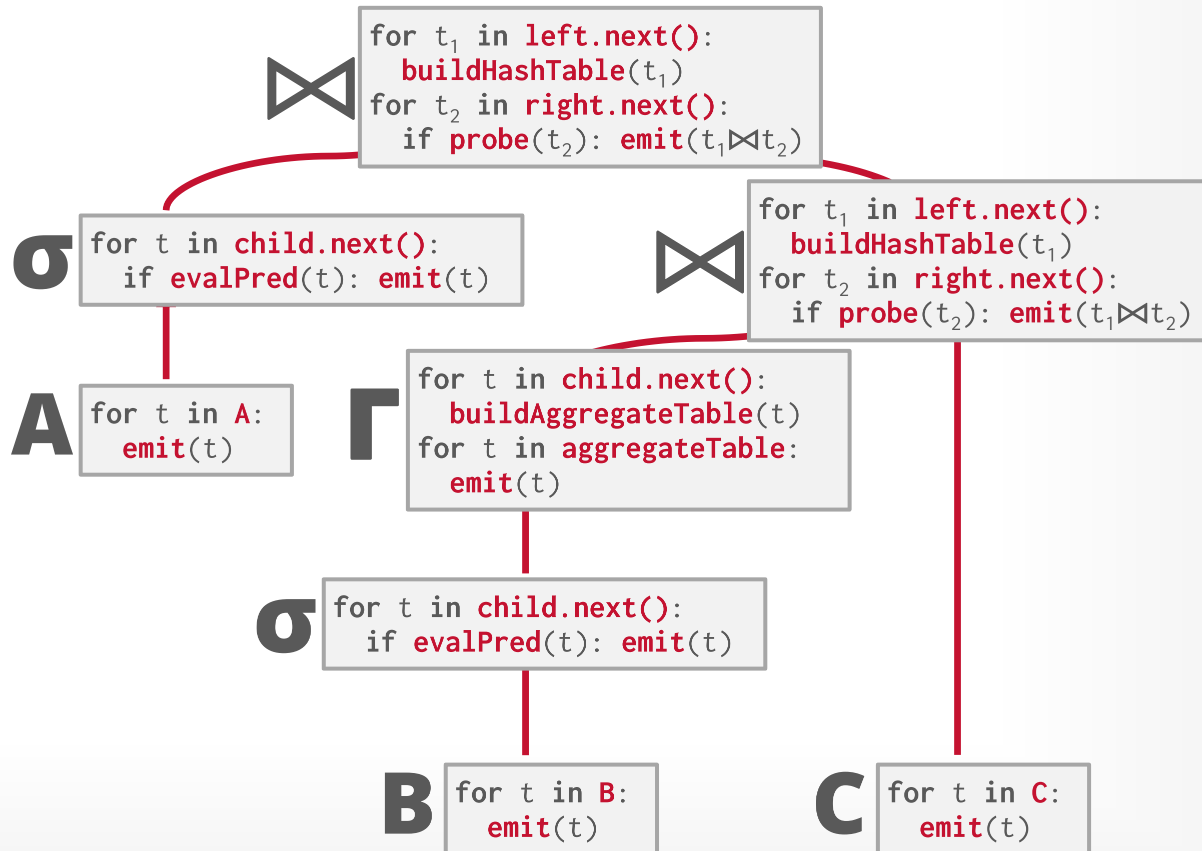
QUERY INTERPRETATION

```
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
```



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
```

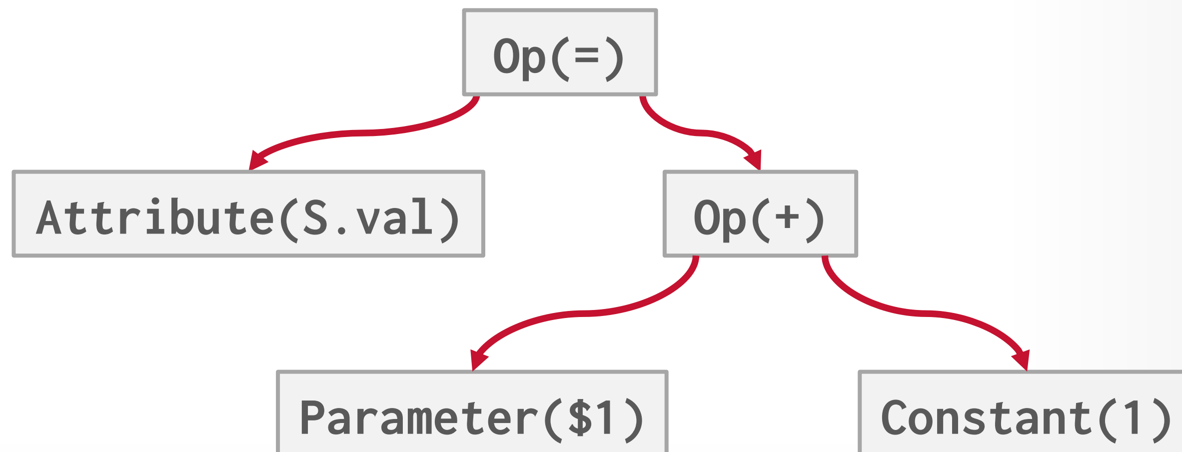


EXPRESSION EVALUATION

```
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

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

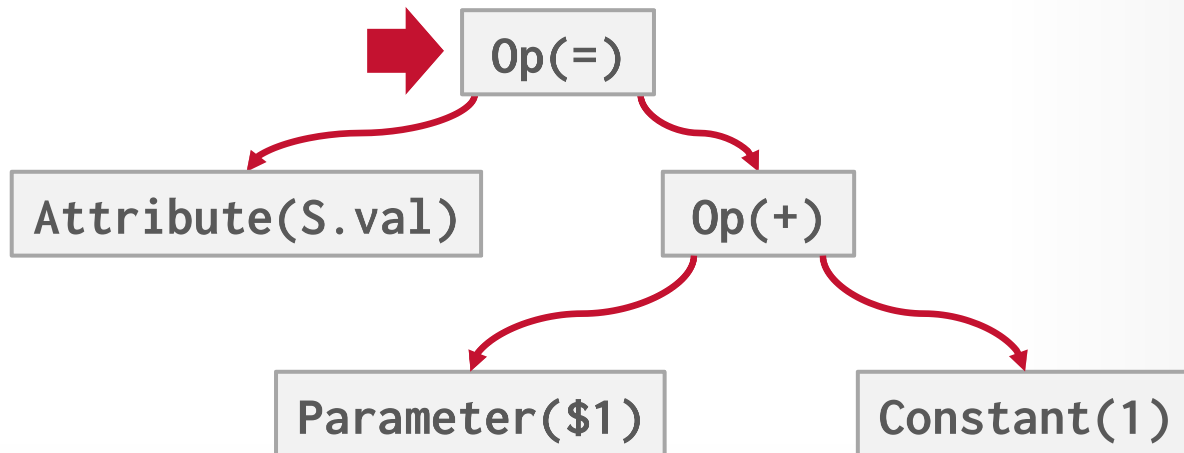


EXPRESSION EVALUATION

```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
```

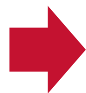
Execution Context

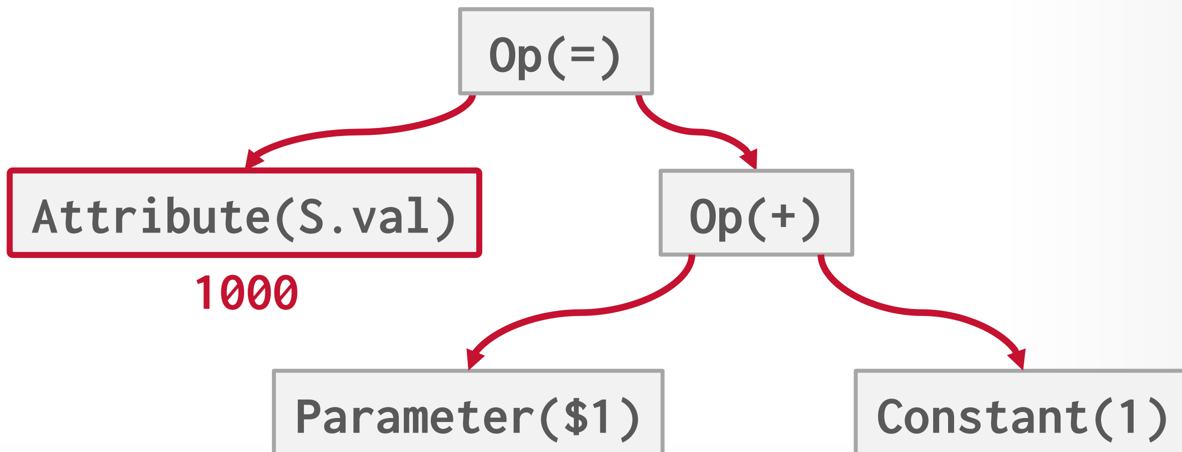
Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	B→(int:id, int:val)



EXPRESSION EVALUATION

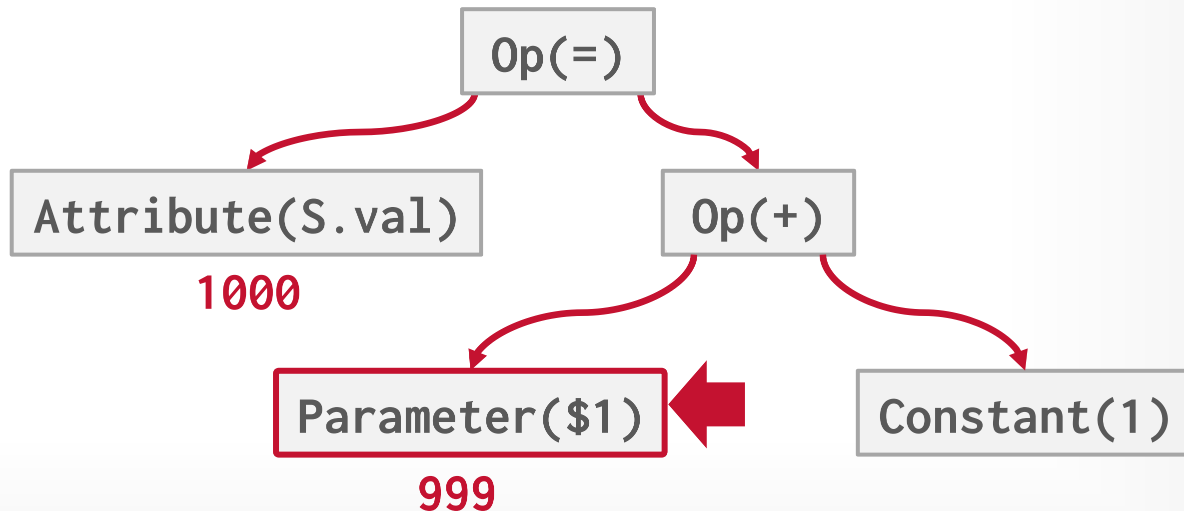
```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```





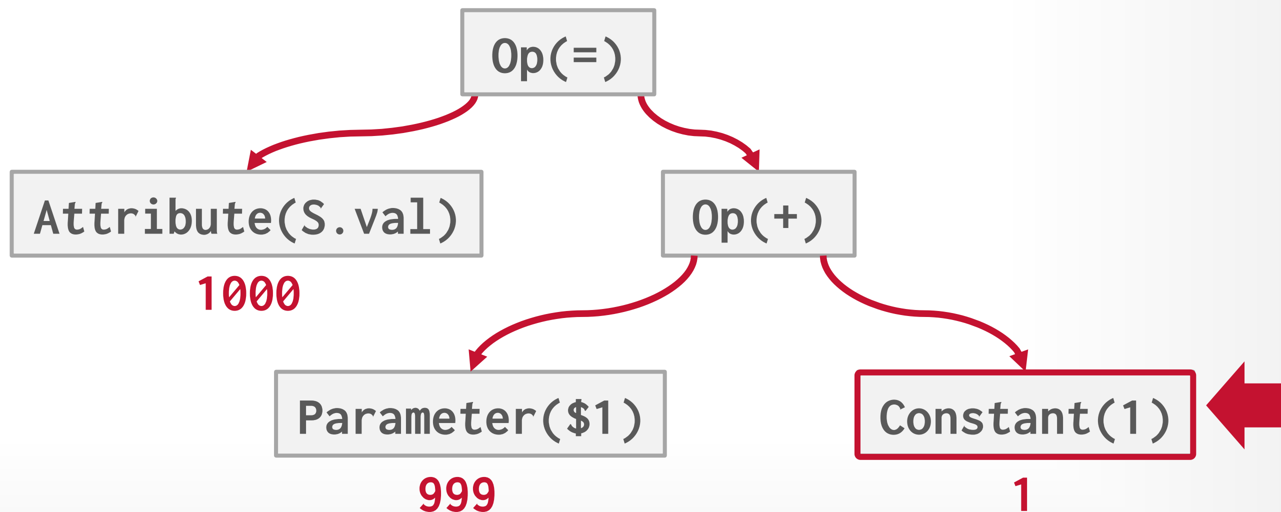
EXPRESSION EVALUATION

```
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

```
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
```

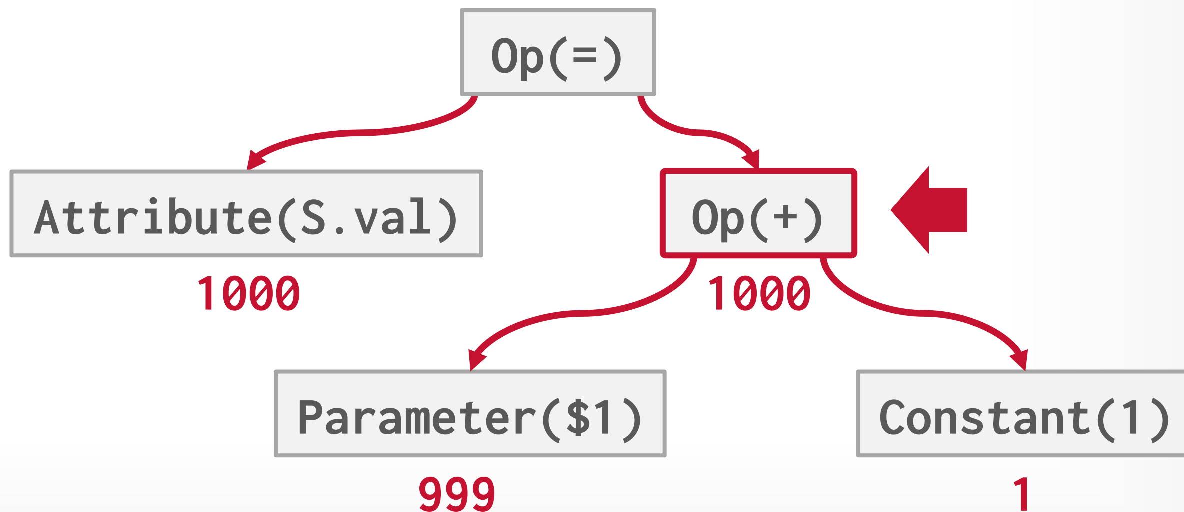


EXPRESSION EVALUATION


```
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
```

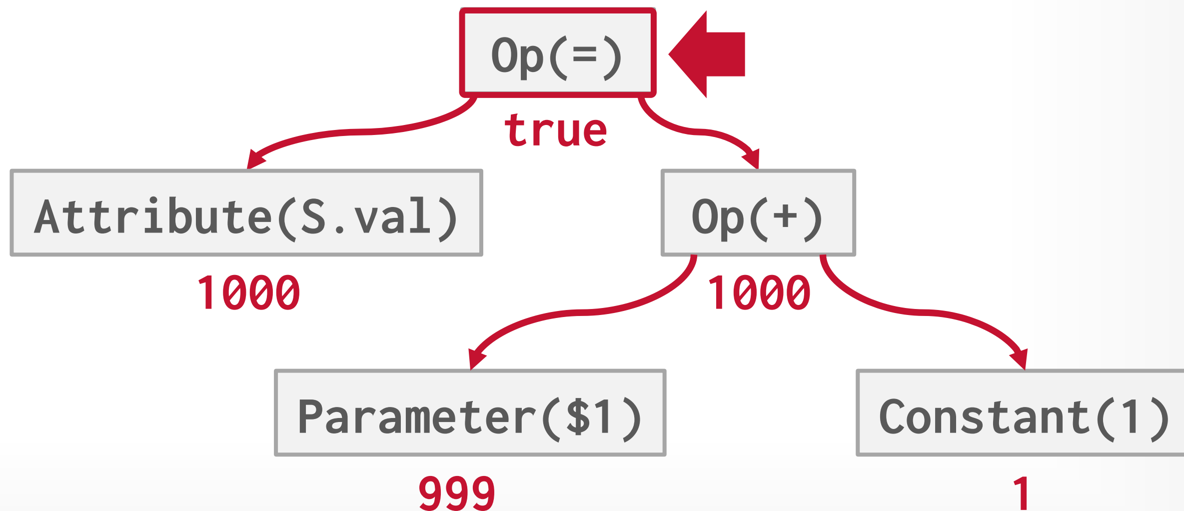
Execution Context

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



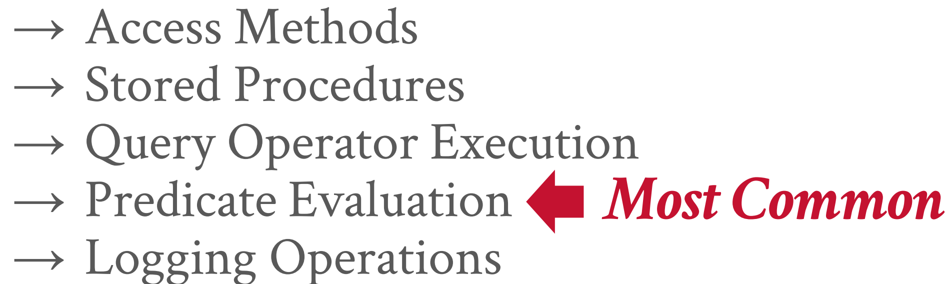
EXPRESSION EVALUATION

```
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
```



CODE SPECIALIZATION

The DBMS generates code for any CPU-intensive task that has a similar execution pattern on different inputs.



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

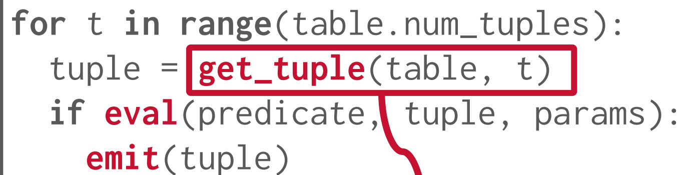
For a given query plan, create a C/C++ program that implements that query's execution.
→ Bake in all the predicates and type conversions.

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.

GENERATING CODE FOR HOLISTIC QUERY EVALUATION
ICDE 2010

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.*

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.*

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.*

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.*

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)
```

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.*

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.*

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)
```

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.

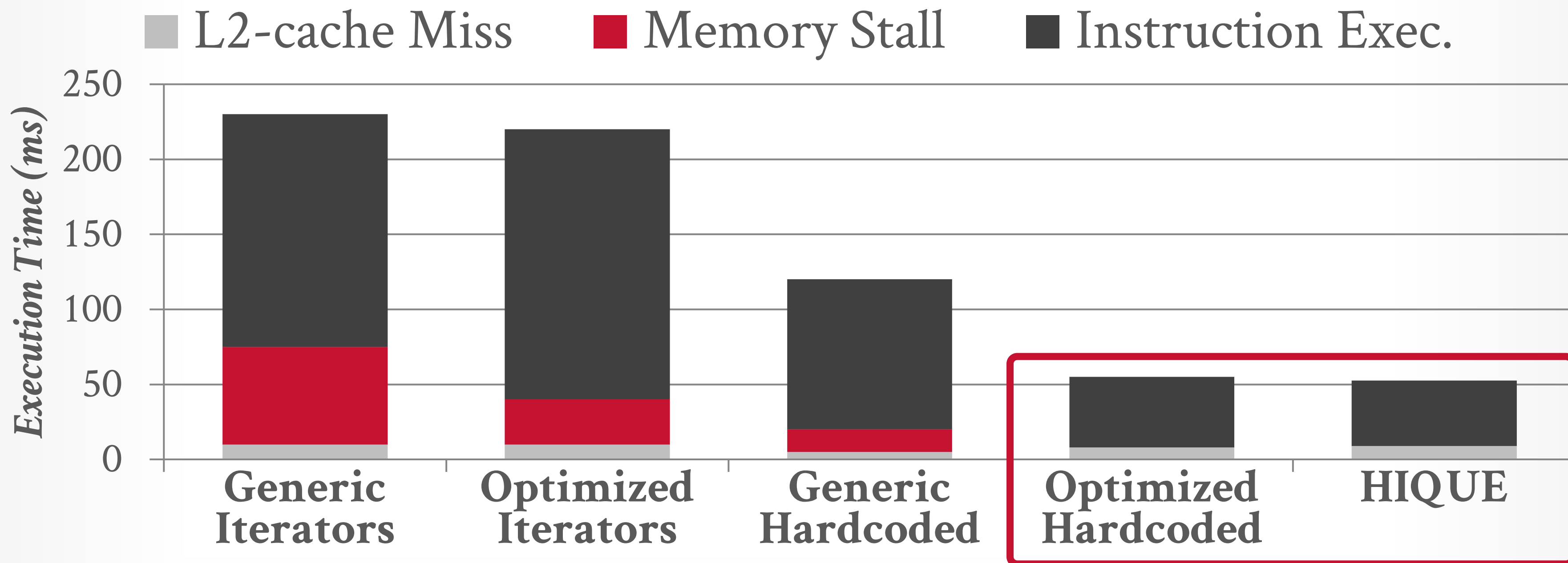
Generic Hardcoded
→ Handwritten code with generic iterators/predicates.

Optimized Hardcoded
→ Direct tuple access with pointer arithmetic.

HIQUE
→ Query-specific specialized code.

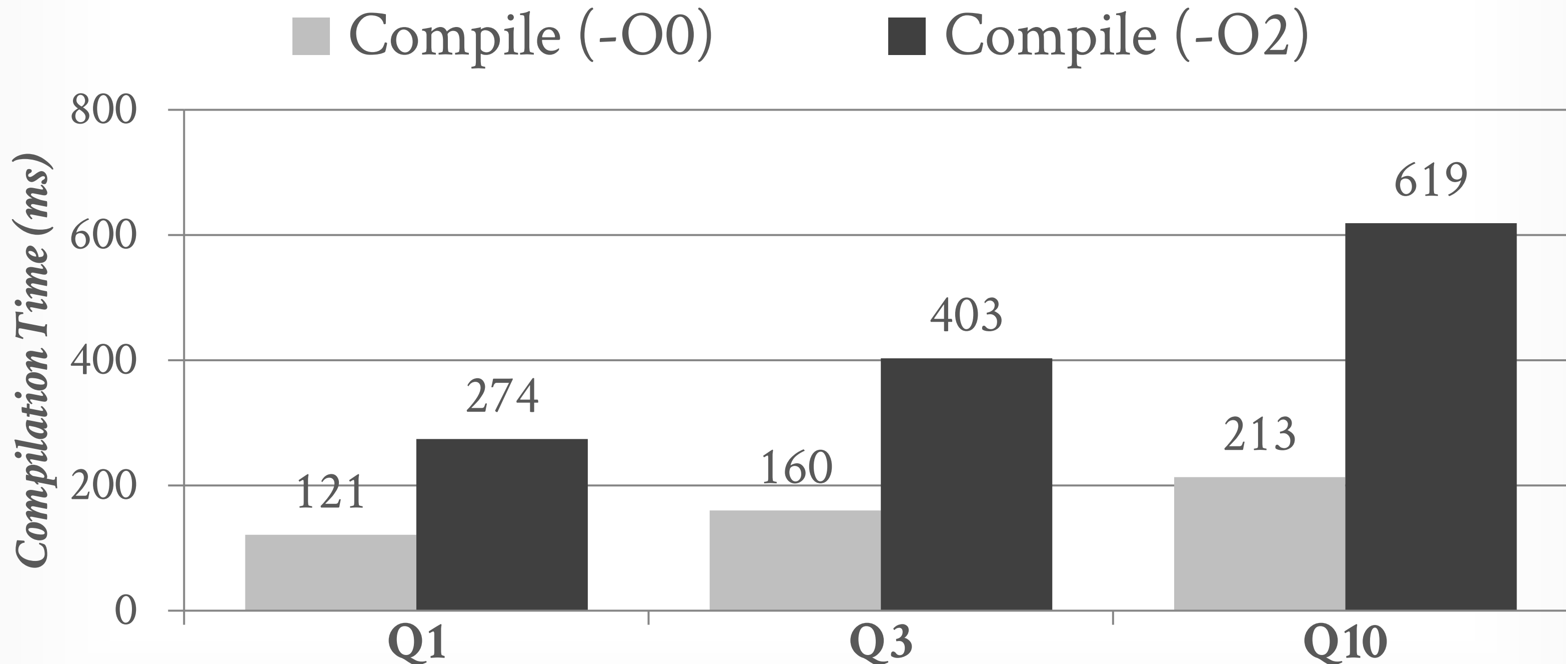
QUERY COMPILATION EVALUATION

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



Source: Konstantinos Krikellas

QUERY COMPILATION COST



Source: Konstantinos Krikellas

OBSERVATION

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

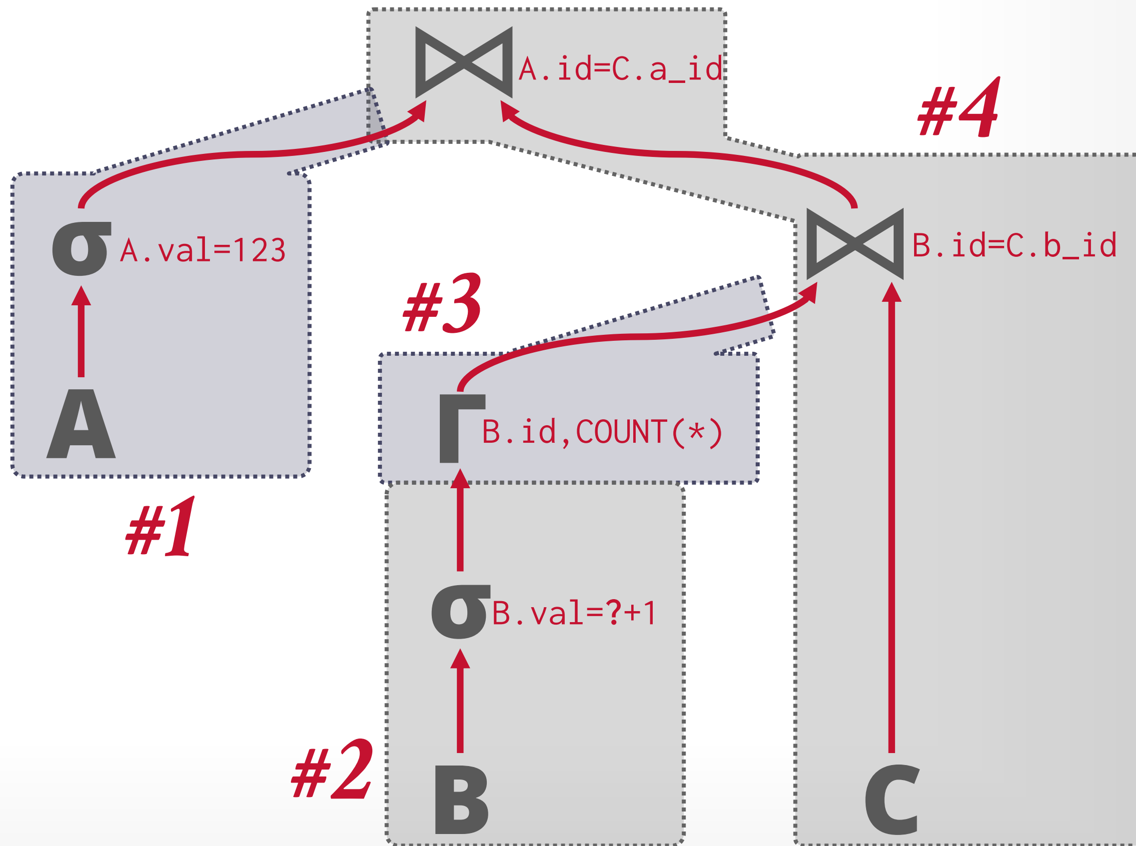
→ Data Centric vs. Operator Centric

EFFICIENTLY COMPILING EFFICIENT QUERY PLANS FOR MODERN HARDWARE
VLDB 2011

PIPELINED OPERATORS

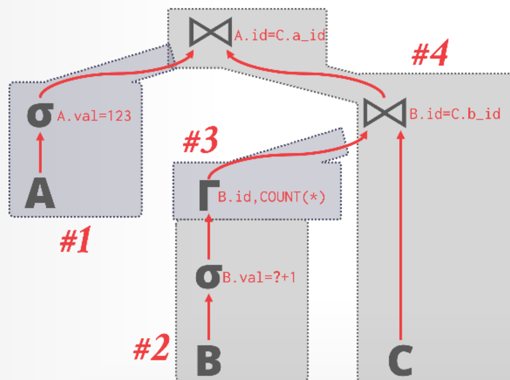
```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Pipeline Boundaries



PUSH-BASED EXECUTION

```
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
```

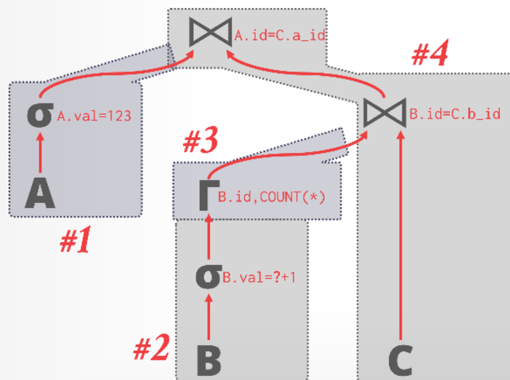


Generated Query Plan

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

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

for t in Γ(B.id):
  Materialize t in HashTable ⋈(B.id=C.b_id)

for t3 in C:
  for t2 in ⋈(B.id=C.b_id):
    for t1 in ⋈(A.id=C.a_id):
      emit(t1⋈t2⋈t3)
```

PUSH-BASED EXECUTION

```
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
```



Generated Query Plan

#1
```
for t in A:
  if t.val == 123:
    Materialize t in HashTable ⋈(A.id=C.a_id)
```

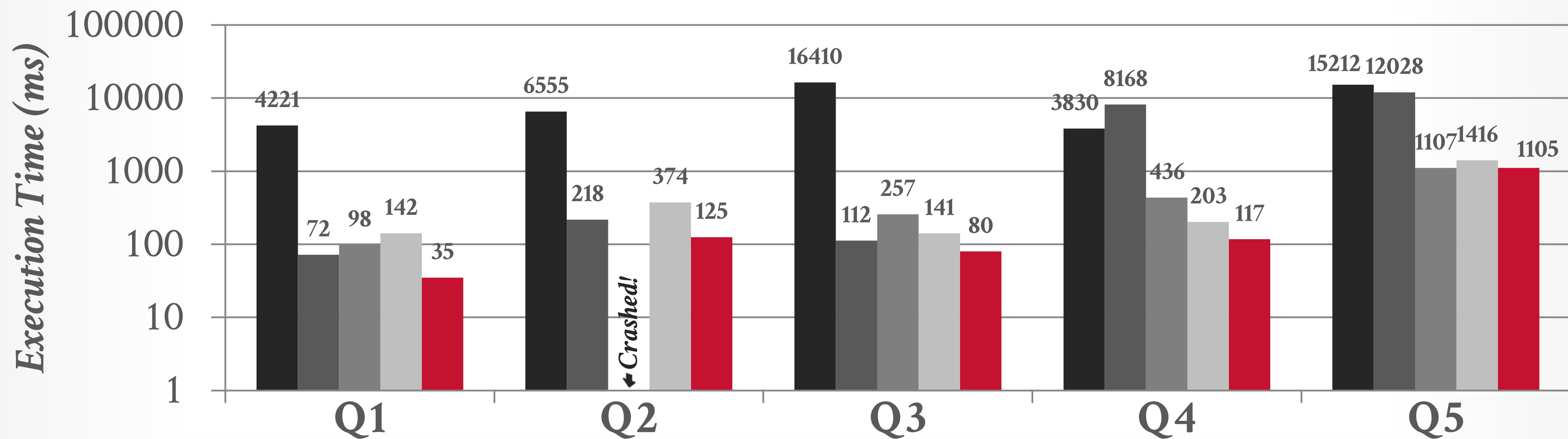
#2
```
for t in B:
  if t.val == <param> + 1:
    Aggregate t in HashTable Γ(B.id)
```

#3
```
for t in Γ(B.id):
  Materialize t in HashTable ⋈(B.id=C.b_id)
```

#4
```
for t3 in C:
  for t2 in ⋈(B.id=C.b_id):
    for t1 in ⋈(A.id=C.a_id):
      emit(t1⋈t2⋈t3)
```

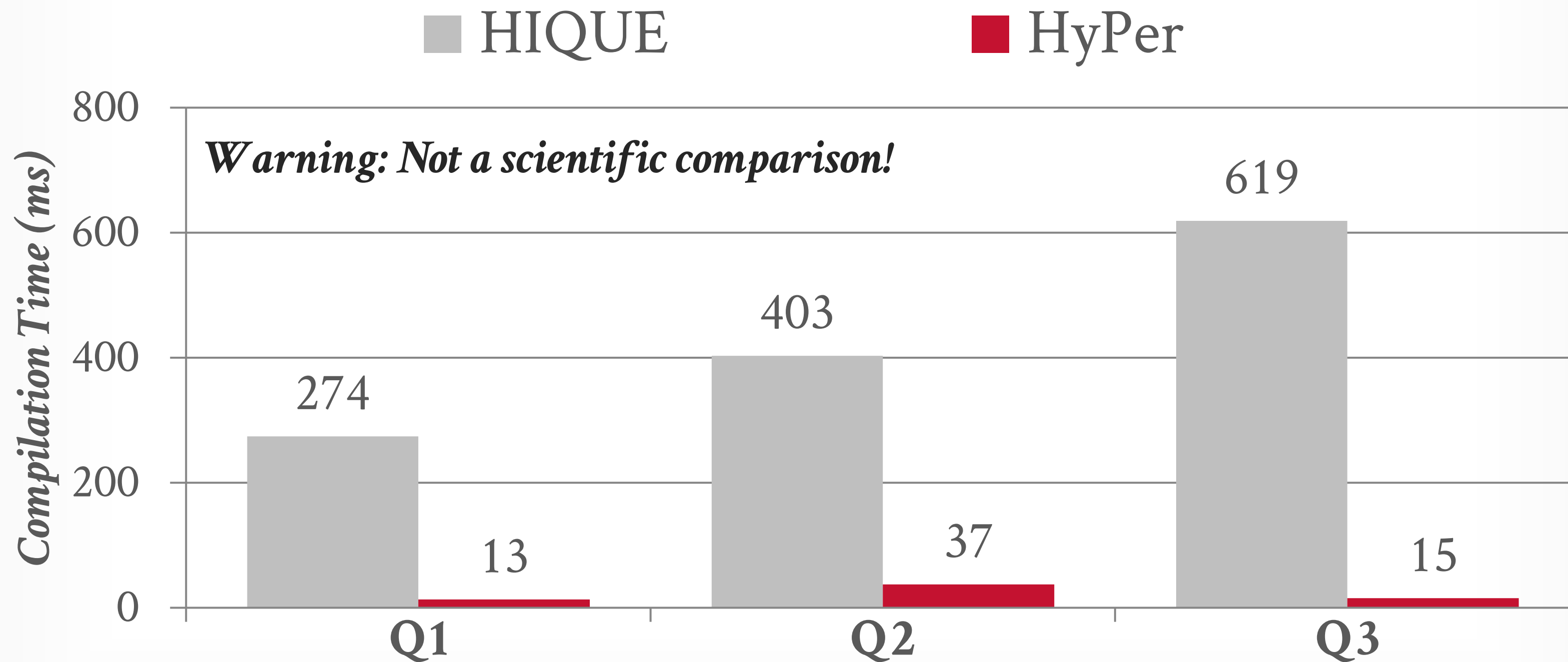
QUERY COMPILATION EVALUATION

Dual Socket Intel Xeon X5770 @ 2.93GHz
TPC-H Queries (Scalefactor=1)



Source: Thomas Neumann

QUERY COMPILATION COST



Source: Konstantinos Krikellas

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.

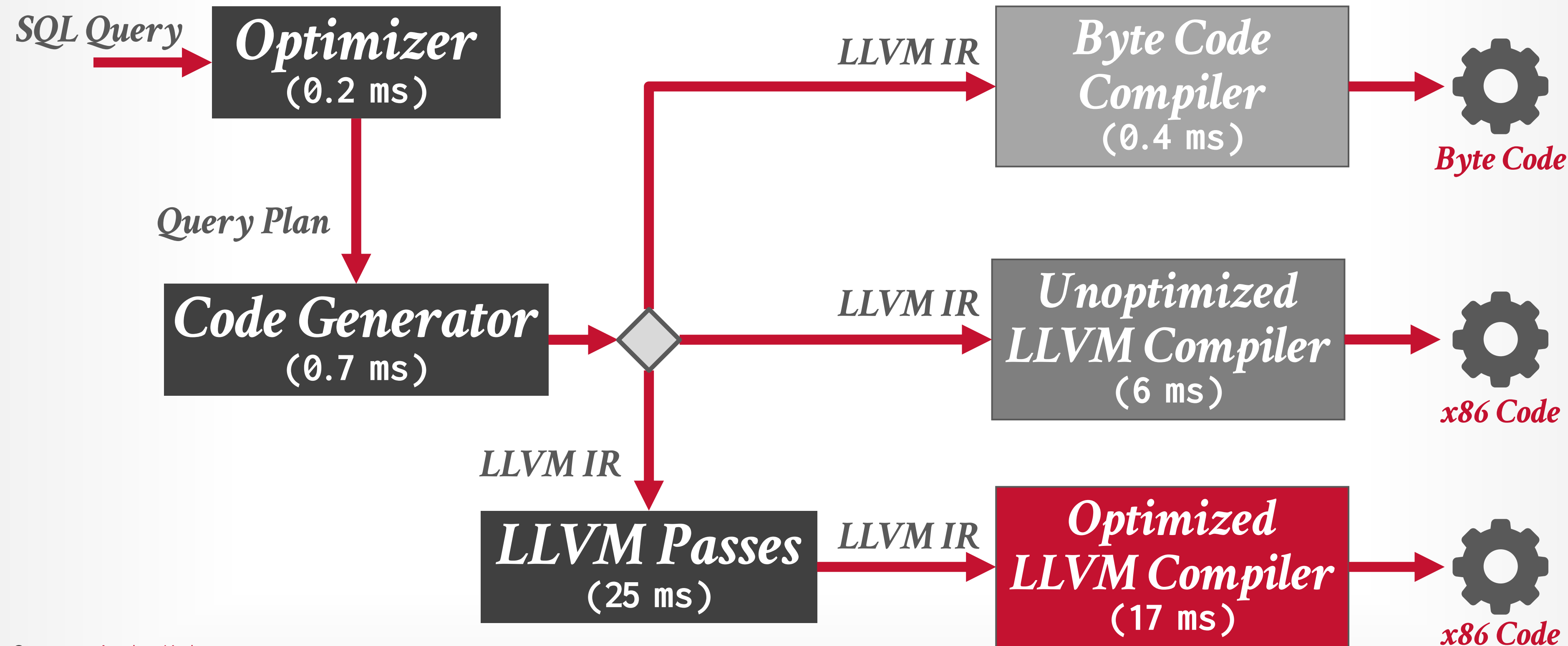
Then the DBMS compiles the query in the background.

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

→ For each morsel, check to see whether the compiled version is available.

ADAPTIVE EXECUTION OF COMPILED QUERIES
ICDE 2018

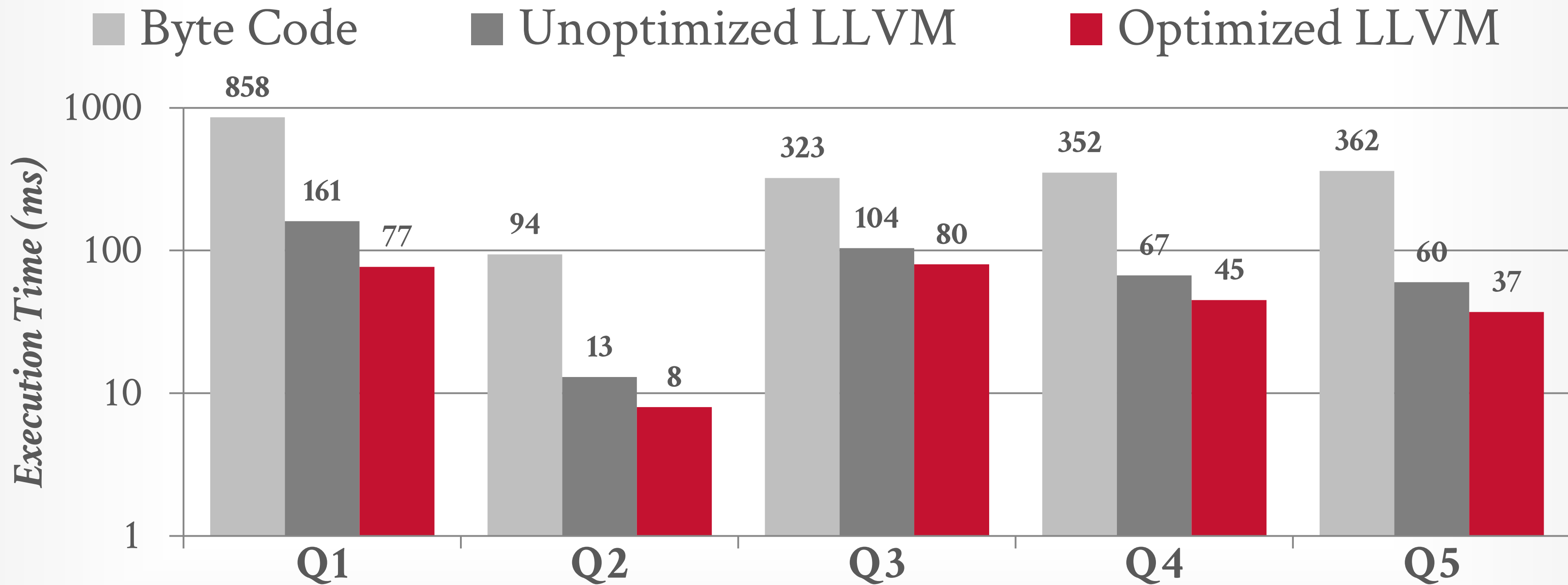
HYPER: ADAPTIVE EXECUTION



Source: Andre Kohn

HYPER: ADAPTIVE EXECUTION

AMD Ryzen 7 1700X @ 3.4GHz (One Thread)
TPC-H Queries



Source: Andre Kohn

REAL-WORLD IMPLEMENTATIONS

Transpilation

Amazon Redshift

Oracle

MemSQL (2016)

Custom

IBM System R

Actian Vector

Microsoft Hekaton

SQLite

TUM HyPer

TUM Umbra

QuestDB

JVM-based

Spark

Neo4j

Splice Machine

Presto / Trino

OrientDB

Tajo

Derby

LLVM-based

SingleStore

VitesseDB

PostgreSQL (2018)

CMU Peloton

CMU NoisePage

TUM LingoDB

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 HISTORY AND EVALUATION OF SYSTEM R
COMMUNICATIONS OF THE ACM 1981

IBM SYSTEM R

A primitive form of code generation an
compilation was used by IBM in 1970s.

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

Technique was abandoned when IBM
and DB2 in the 1980s:

→ High cost of external function calls
→ Poor portability
→ Software engineer complications

The Compilation Approach

Perhaps the most important decision in the design of the RDS was inspired by R. Lorie's observation, in early 1976, that it is possible to compile very high-level SQL statements into compact, efficient routines in System/370 machine language [42]. Lorie was able to demonstrate that SQL statements of arbitrary complexity could be decomposed into a relatively small collection of machine-language "fragments," and that an optimizing compiler could assemble these code fragments from a library to form a specially tailored routine for processing a given SQL statement. This technique had a very dramatic effect on our ability to support application programs for transaction processing. In System R, a

A HISTORY AND EVALUATION OF SYSTEM R
COMMUNICATIONS OF THE ACM 1981

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.

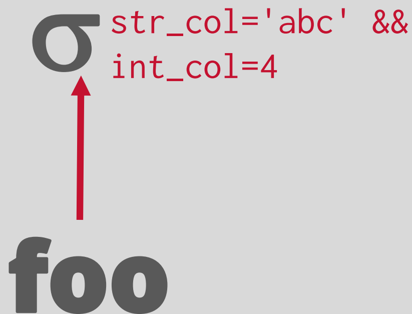
→ Function calls are amortized over multiple tuples.

→ The output of a primitive are the offsets of tuples that

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

VECTORWISE: PRECOMPILED PRIMITIVES

```sql
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);
}
```

AMAZON REDSHIFT

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.

AMAZON REDSHIFT RE-INVENTED
SIGMOD 2022

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.

COMPILATION IN THE MICROSOFT SQL SERVER HEKATON ENGINE
IEEE DATA ENGINEERING BULLETIN 2011

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   Start at 4
1     Add            2     2     1                    0   r[1]=r[2]+r[2]
2     ResultRow      1     1     0                    0   output=r[1]
3     Halt           0     0     0                    0
4     Integer        1     2     0                    0   r[2]=1
5     Goto           0     1     0                    0
Run Time: real 0.000 user 0.000185 sys 0.000000
```

Source: Richard Hipp

TUM UMBRA

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.

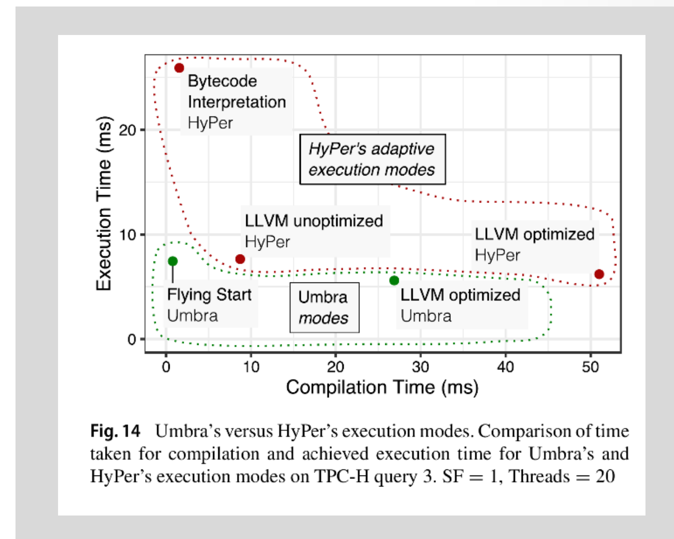


Fig. 14 Umbra's versus HyPer's execution modes. Comparison of time taken for compilation and achieved execution time for Umbra's and HyPer's execution modes on TPC-H query 3. SF = 1, Threads = 20

They also wrote their own debugger!

TIDY TUPLES AND FLYING START: FAST COMPILATION AND FAST EXECUTION OF RELATIONAL QUERIES IN UMBRA
VLDB JOURNAL 2021

TUM UMB

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!

On Another Level: How to Debug Compiling Query Engines

Timo Kersten
kersten@in.tum.de
Technical University of Munich

Thomas Neumann
neumann@in.tum.de
Technical University of Munich

ABSTRACT

Compilation-based query engines generate and compile code at runtime, which is then run to get the query result. In this process there are two levels of source code involved: The code of the code generator itself and the code that is generated at runtime. This can make debugging quite indirect, as a fault in the generated code was caused by an error in the generator. To find the error, we have to look at both, the generated code and the code that generated it.

Current debugging technology is not equipped to handle this situation. For example, GNU's gdb only offers facilities to inspect one source line, but not multiple source levels. Also, current debuggers are not able to reconstruct additional program state for further source levels, thus, context is missing during debugging.

In this paper, we show how to build a multi-level debugger for generated queries that solves these issues. We propose to use a time-travelling debugger to provide context information for compile-time and runtime, thus providing full interactive debugging capabilities for every source level. We also present how to build such a debugger with low engineering effort by combining existing tool chains.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging.**

KEYWORDS

Relational Query Execution, Code Generation, Debugging

ACM Reference Format:

Timo Kersten and Thomas Neumann. 2020. On Another Level: How to Debug Compiling Query Engines. In *Workshop on Testing Database Systems (DBTest'20), June 19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3395032.3395321

1 INTRODUCTION

With the advent of in-memory databases, high-bandwidth solid state drives, and recently also persistent memory [11], high-performance relational query execution engines compile machine code for query execution. This approach creates optimal code for each query and thus makes best use of available computing resources [12]. Consequently, code generating execution engines are able to make the most of the large available bandwidth.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DBTest'20, June 19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8001-0/20/06…$15.00
https://doi.org/10.1145/3395032.3395321

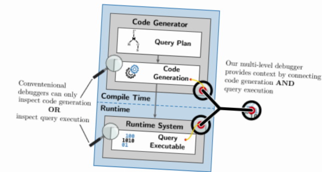


Figure 1: Compiling relational engines process queries in two steps: *Code generation and execution. Conventional debuggers can only attach to one step, so that debugging execution misses lots of context information. Our multi-level debugger provides this context.*

Query execution in a compiling query engine is done in a two-step process (cf. Figure 1). First, the engine generates code for the query plan. Second, the machine's processors execute this code to compute the query result [15]. For the developer of a compiling engine this two-step process can become a challenge. When, during development, they find their computation results are wrong, they need *debugging tools* to efficiently triangulate the cause of the fault.

Conventional debuggers support the search of errors by allowing the developer to stop the execution at any point. The developer can then inspect the program state, view the value of variables, explore data structures, and examine the call-stack to decide whether the observed behavior is as expected or already affected by an error. To make this process efficient, the debugger should show the developer a full view of the program state in the source language and the format that the developer wrote it. In other words, the debugger should present the state in terms the developer is familiar with.

In a compiling query engine, however, this integrated experience is not possible with a regular debugger. A compiling engine splits the query execution into the two phases shown in Figure 1: *Compile time*, which generates code for a query plan and compiles it to machine instructions, and *runtime*, which runs the machine instructions to produce the query result. To debug this two-level setup, most toolchains already offer the means to step through either the code generator or the runtime code. However, the *link* between the generated code and the source code that generated it, is *missing*. Without the link the developer is missing most of the query context.

Currently, there are two limitations that cause this disconnect: First, current debuggers are not built for this kind of debugging. GDB, for example, supports only to *stop at one position* in the machine code and map that position to one source location. There is currently no support to handle a second source location that

TIDY TUPLES AND FLYING START: FAST COMPILATION AND FAST EXECUTION OF RELATIONAL QUERIES IN UMBRA
VLDB JOURNAL 2021

JAVA DATABASES

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.

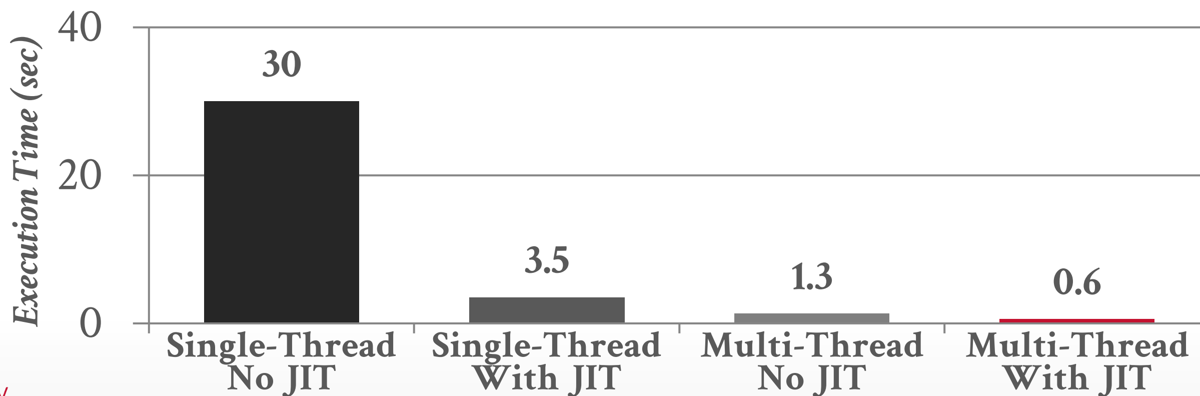
Databricks abandoned this approach with their new Photon engine in late 2010s.

SPARK SQL: RELATIONAL DATA PROCESSING IN SPARK
SIGMOD 2015

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.



Source: Andrey Pechkurov

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.

SINGLESTORE (2016-PRESENT)

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

POSTGRESQL

Adde
pred
→ R
co

Aut
into

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)

EDB POSTGRES

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

CMU NOISEPAGE (2019)

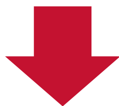
SingleStore-style conversion of query plans into a database-oriented DSL.

Then compile the DSL into opcodes.

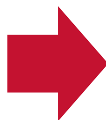
HyPer-style interpretation of opcodes while compilation occurs in the background with LLVM.

CMU NOI

```
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
}
```



```
Function 0 <main>:
[3/4587]
  Frame size 8512 bytes (1 parameter, 20 locals)
    param     hiddenRv:  offset=0       size=8      align=8     type=*int32
    local          ret:  offset=8       size=4      align=4     type=int32
    local   table_iter:  offset=16      size=8312   align=8     type=tpl::sql::TableVectorIterator
    local          vpi:  offset=8328    size=8      align=8     type=*tpl::sql::VectorProjectionIterator
    local         tmp1:  offset=8336    size=1      align=1     type=bool
    local          row:  offset=8344    size=64     align=8     type=struct{Integer,Integer,Integer,Integer}
    local         tmp2:  offset=8408    size=1      align=1     type=bool
    local         tmp3:  offset=8416    size=8      align=8     type=*Integer
    local         tmp4:  offset=8424    size=8      align=8     type=*Integer
    local         tmp5:  offset=8432    size=8      align=8     type=*Integer
    local         tmp6:  offset=8440    size=8      align=8     type=*Integer
    local         tmp7:  offset=8448    size=1      align=1     type=bool
    local         tmp8:  offset=8449    size=2      align=1     type=Boolean
    local         tmp9:  offset=8456    size=16     align=8     type=Integer
    local        tmp10:  offset=8472    size=4      align=4     type=int32
    local        tmp11:  offset=8476    size=2      align=1     type=Boolean
    local        tmp12:  offset=8480    size=8      align=8     type=*Integer
    local        tmp13:  offset=8488    size=16     align=8     type=Integer
    local        tmp14:  offset=8504    size=4      align=4     type=int32
    local        tmp15:  offset=8508    size=4      align=4     type=int32

  0x00000000    AssignImm4
  0x0000000c    TableVectorIteratorInit
  0x00000016    TableVectorIteratorGetVPI
  0x00000022    TableVectorIteratorNext
  0x0000002e    JumpIfFalse
  0x0000003a    VPIHasNext
  0x00000046    JumpIfFalse
  0x00000052    Lea
  0x00000062    VPIGetInteger
  0x00000072    Lea
  0x00000082    VPIGetInteger
  0x00000092    Lea
  0x000000a2    VPIGetInteger
  0x000000b2    Lea
  0x000000c2    VPIGetInteger
  0x000000d2    AssignImm4
  0x000000de    InitInteger
  0x000000ea    GreaterThanEqualInteger
  0x000000fa    ForceBoolTruth
  0x00000106    JumpIfFalse
```

Source: Prashanth Menon

CMU NOISEPAGE (2019)

```
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
}
```

```
Function 0 <main>:
[3/4587]
  Frame size 8512 bytes (1 parameter, 20 locals)
    param    hiddenRv:  offset=0      size=8     align=8   type=*int32
    local         ret:  offset=8      size=4     align=4   type=int32
    local  table_iter:  offset=16     size=8312  align=8   type=tpl::sql::TableVectorIterator
    local         vpi:  offset=8328   size=8     align=8   type=*tpl::sql::VectorProjectionIterator
    local        tmp1:  offset=8336   size=1     align=1   type=bool
    local         row:  offset=8344   size=64    align=8   type=struct{Integer,Integer,Integer,Integer}
    local        tmp2:  offset=8408   size=1     align=1   type=bool
    local        tmp3:  offset=8416   size=8     align=8   type=*Integer
    local        tmp4:  offset=8424   size=8     align=8   type=*Integer
    local        tmp5:  offset=8432   size=8     align=8   type=*Integer
    local        tmp6:  offset=8440   size=8     align=8   type=*Integer
    local        tmp7:  offset=8448   size=1     align=1   type=bool
    local        tmp8:  offset=8449   size=2     align=1   type=Boolean
    local        tmp9:  offset=8456   size=16    align=8   type=Integer
    local       tmp10:  offset=8472   size=4     align=4   type=int32
    local       tmp11:  offset=8476   size=2     align=1   type=Boolean
    local       tmp12:  offset=8480   size=8     align=8   type=*Integer
    local       tmp13:  offset=8488   size=16    align=8   type=Integer
    local       tmp14:  offset=8504   size=4     align=4   type=int32
    local       tmp15:  offset=8508   size=4     align=4   type=int32

  0x00000000    AssignImm4
  0x0000000c    TableVectorIteratorInit
  0x00000016    TableVectorIteratorGetVPI
  0x00000022    TableVectorIteratorNext
  0x0000002e    JumpIfFalse
  0x0000003a    VPIHasNext
  0x00000046    JumpIfFalse
  0x00000052    Lea
  0x00000062    VPIGetInteger
  0x00000072    Lea
  0x00000082    VPIGetInteger
  0x00000092    Lea
  0x000000a2    VPIGetInteger
  0x000000b2    Lea
  0x000000c2    VPIGetInteger
  0x000000d2    AssignImm4
  0x000000de    InitInteger
  0x000000ea    GreaterThanEqualInteger
  0x000000fa    ForceBoolTruth
  0x00000106    JumpIfFalse
  0x00000112    Lea
  0x00000122    AssignImm4
  0x0000012e    InitInteger
  0x0000013a    LessThanInteger
  0x0000014a    ForceBoolTruth
  0x00000156    JumpIfFalse
  0x00000162    AssignImm4
  0x0000016e    Add_i32
  0x0000017e    VPIAdvance
  0x00000186    Jump
  0x0000018e    VPIReset
  0x00000196    Jump
  0x0000019e    TableVectorIteratorClose
  0x000001a6    Assign4
  0x000001b2    Return
```

Interpreter

Optimized LLVM Compiler

x86 Code

Source: Prashanth Menon

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)

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. 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

TPC

Q1: Fixed-point ar

Q6: Selective filter

Q3: Join (build: 14

Q9: Join (build: 32

Q18: High-cardina

cmu-db / **benchbase** Public

Notifications | Fork 98 | Star 230

Code | Issues 28 | Pull requests 19 | Actions | Projects | Wiki | Security | Insights

main

benchbase / src / main / java / com / oltpbenchmark / benchmarks / tpch / **procedures** /

Go to file

TheoVanderkooy Fix bugs with TPCH implementation (#204) ...

on Jul 7, 2022 **History**

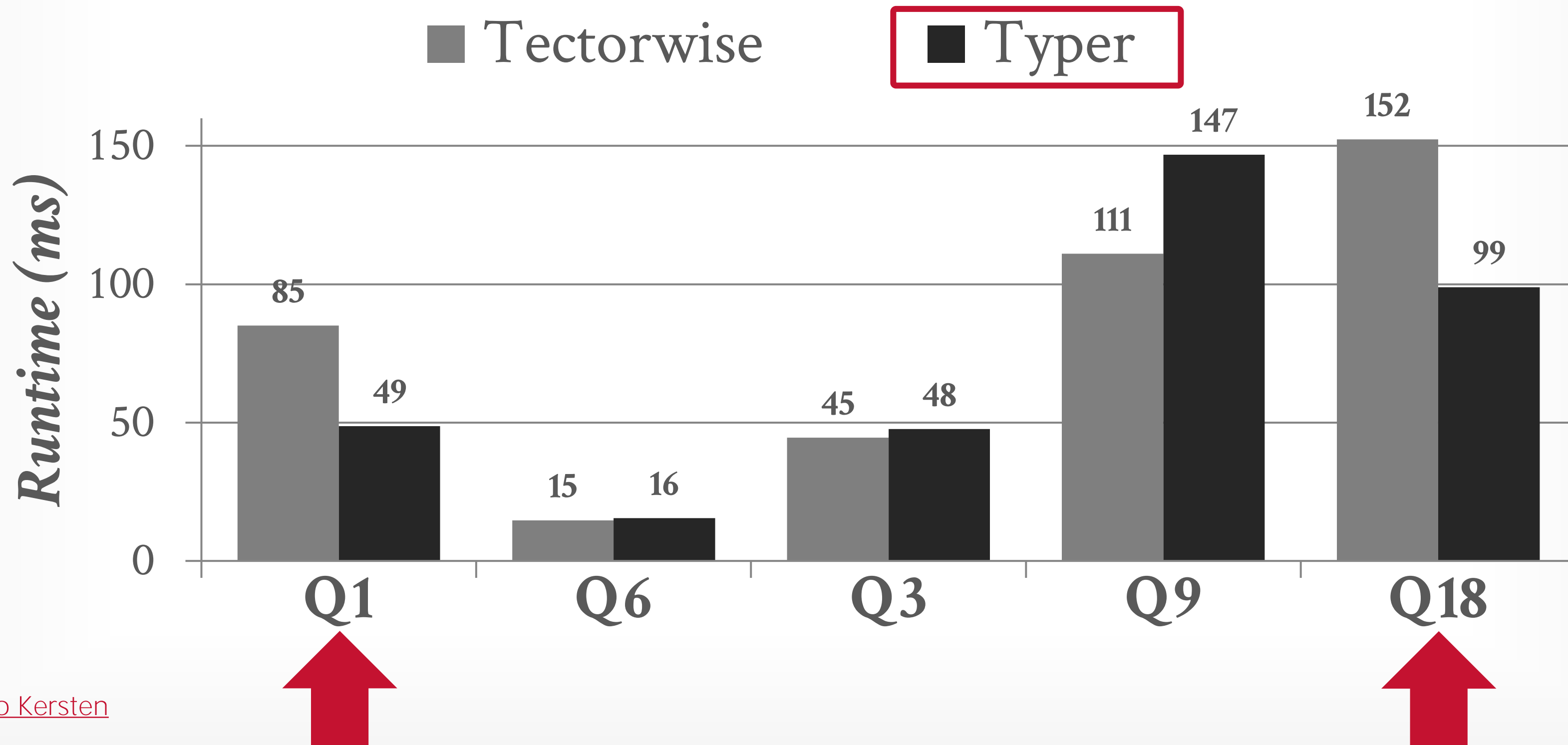
..

File	Last commit	Date
GenericQuery.java	Fix bugs with TPCH implementation (#204)	
Q1.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q10.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q11.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q12.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q13.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q14.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q15.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q16.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q17.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q18.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q19.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q2.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q20.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q21.java	Fix bugs with TPCH implementation (#204)	7 months ago
Q22.java	Fix bugs with TPCH implementation (#204)	7 months ago
		7 months ago

TPC-H ANALYZED: HIDDEN MESSAGES AND LESSONS L
FROM AN INFLUENTIAL BENCHMARK
TPCTC 2013

SINGLE-THREADED PERFORMANCE

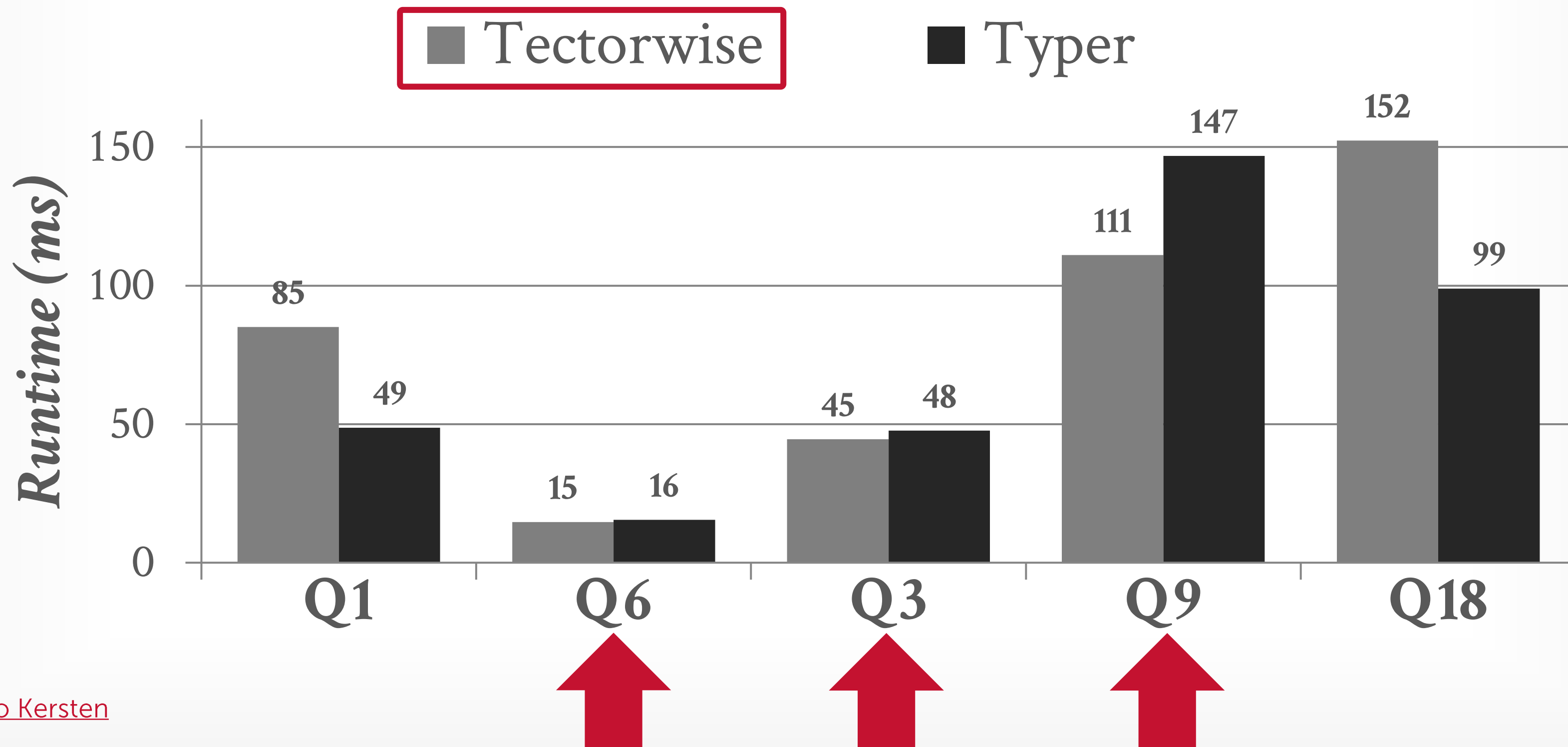
Intel Core i9-7900X (10 cores × 2HT)
TPC-H Queries (Scalefactor=1)



Source: Timo Kersten

SINGLE-THREADED PERFORMANCE

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



Source: Timo Kersten

SINGLE-THREADED PERFORMANCE

		Runtime	*Cycles*	*IPC*	*Instr.*	*L1 Miss*	*LLC Miss*	*Br. Miss*
Q1	TWise	85	59	2.8	162	2.0	0.57	0.03
	Typer	48	34	2.0	68	0.6	0.57	0.01
Q6	TWise	15	11	1.4	15	0.2	0.29	0.01
	Typer	16	11	1.8	20	0.3	0.35	0.06
Q3	TWise	45	24	1.8	42	0.9	0.16	0.08
	Typer	48	25	0.8	21	0.5	0.16	0.27
Q9	TWise	111	56	1.3	76	2.1	0.47	0.39
	Typer	147	74	0.6	42	1.7	0.46	0.34
Q18	TWise	152	48	2.1	102	1.9	0.18	0.37
	Typer	99	30	1.6	46	0.8	0.19	0.16

MAIN FINDINGS

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!