Lecture #14

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Query Compilation & Code Generation

@Andy_Pavlo // 15-721 // Spring 2020

ADMINISTRIVIA

Project #2 Checkpoint: Sunday March 8th

Project #2 Final: Sunday March 15th

Project #3 will be announced next class.

TODAY'S AGENDA

Background
Code Generation / Transpilation
JIT Compilation (LLVM)
Real-world Implementations

HEKATON REMARK

After switching to an in-memory DBMS, 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

OBSERVATION

One way to achieve such a 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…

EXAMPLE DATABASE

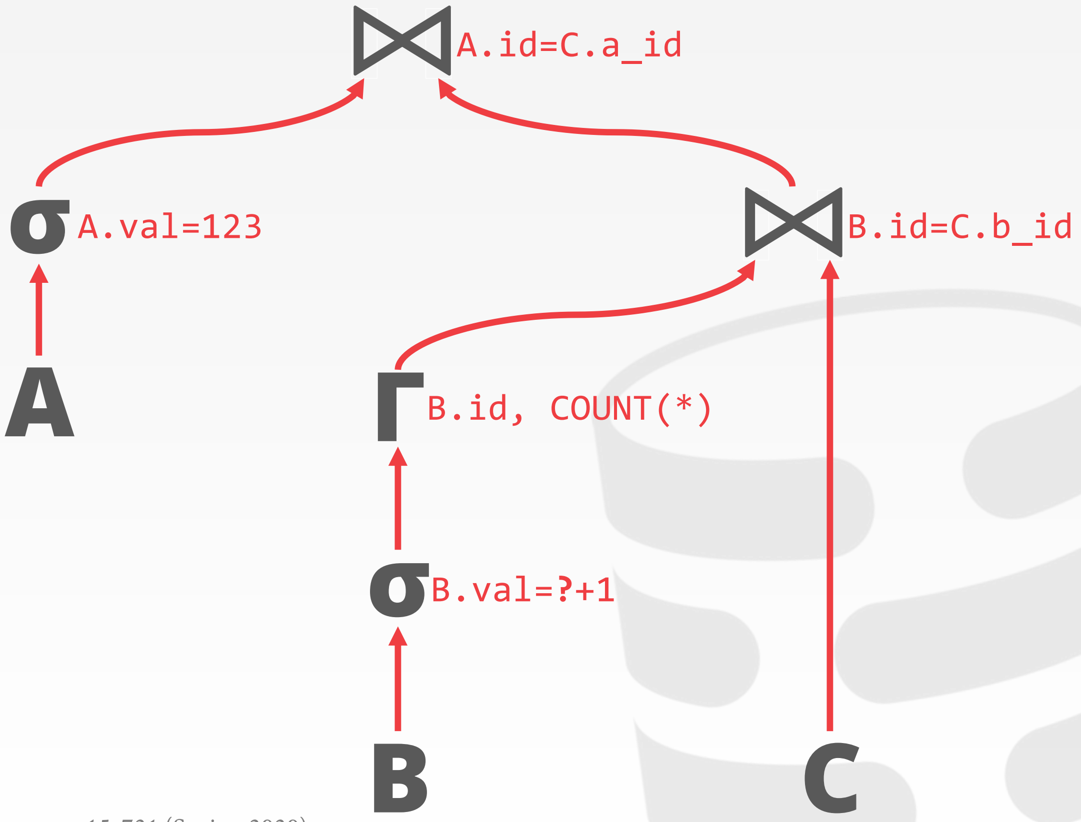
```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT
);
```

```
CREATE TABLE B (
  id INT PRIMARY KEY,
  val INT
);
```

```
CREATE TABLE C (
  a_id INT REFERENCES A(id),
  b_id INT REFERENCES B(id),
  PRIMARY KEY (a_id, b_id)
);
```

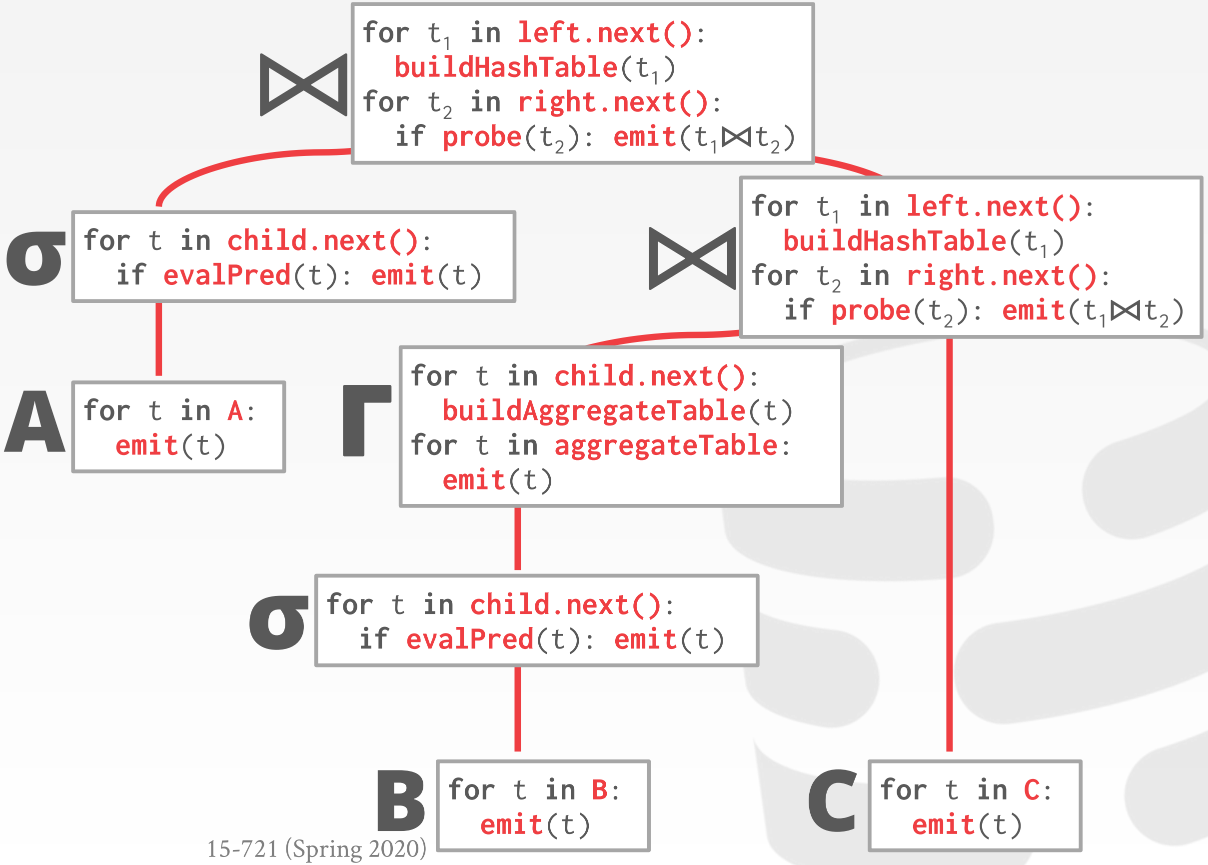
QUERY INTERPRETATION

```
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
```



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
```

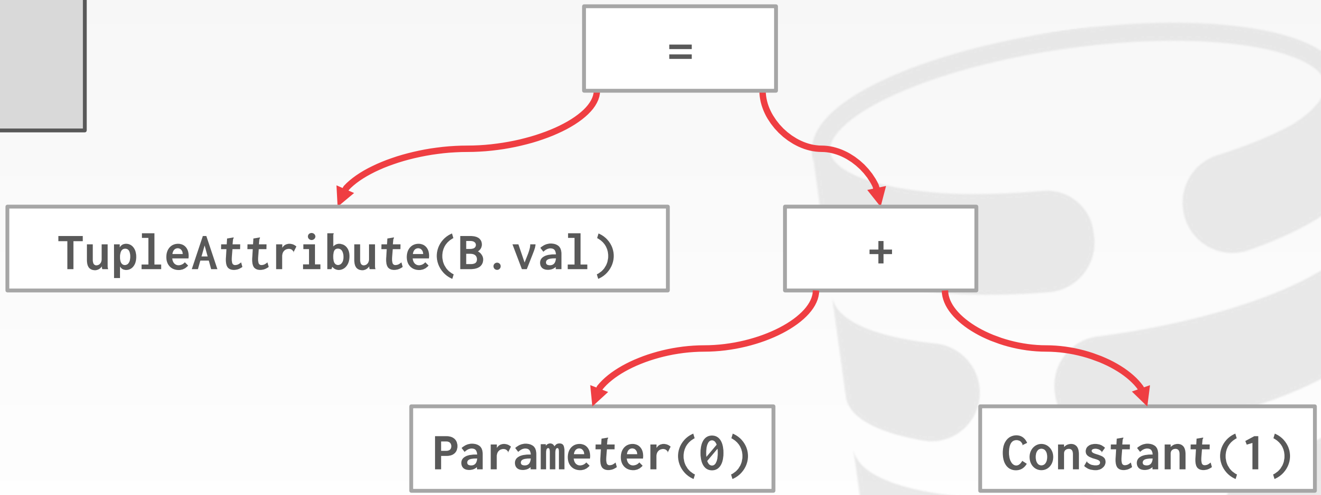


PREDICATE INTERPRETATION

```
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
```

Execution Context

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

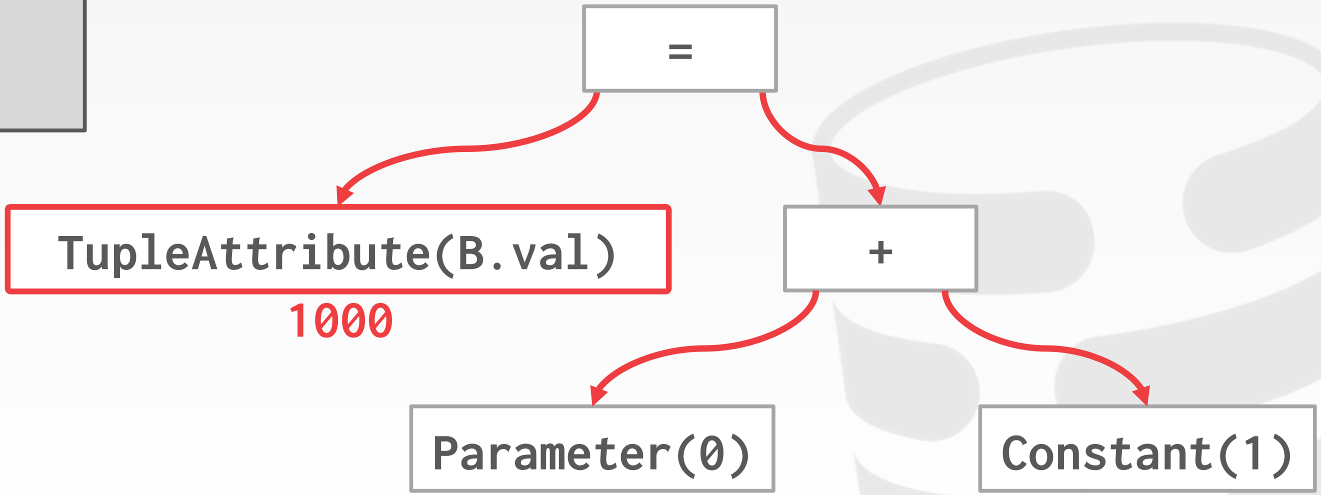


PREDICATE INTERPRETATION

```
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
```

Execution Context

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

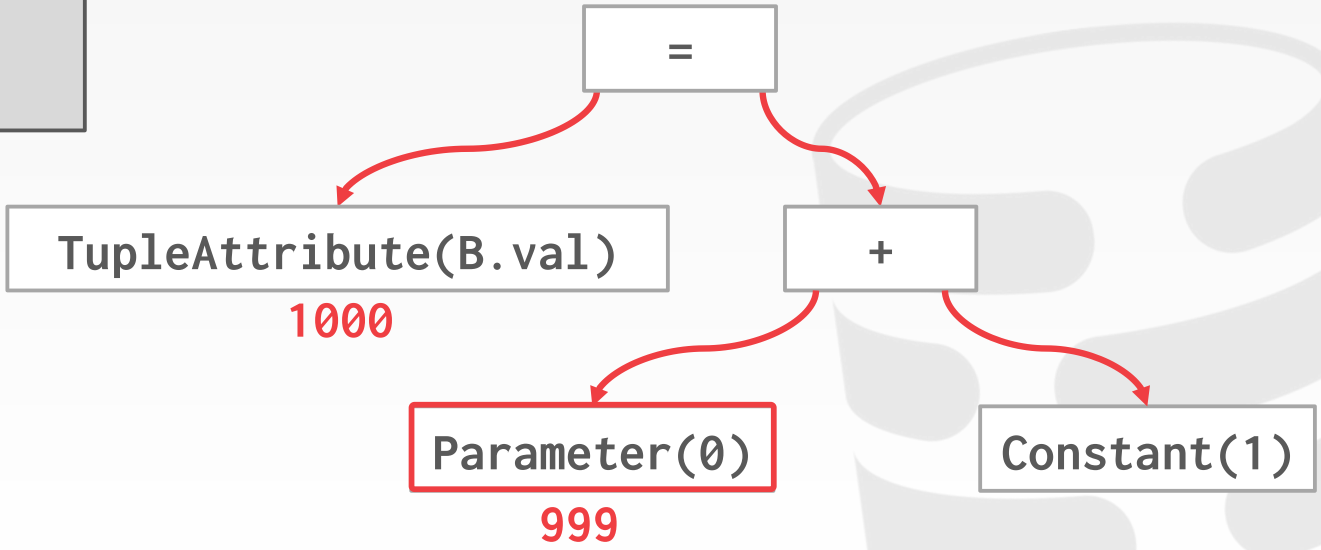


PREDICATE INTERPRETATION

```
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
```

Execution Context

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

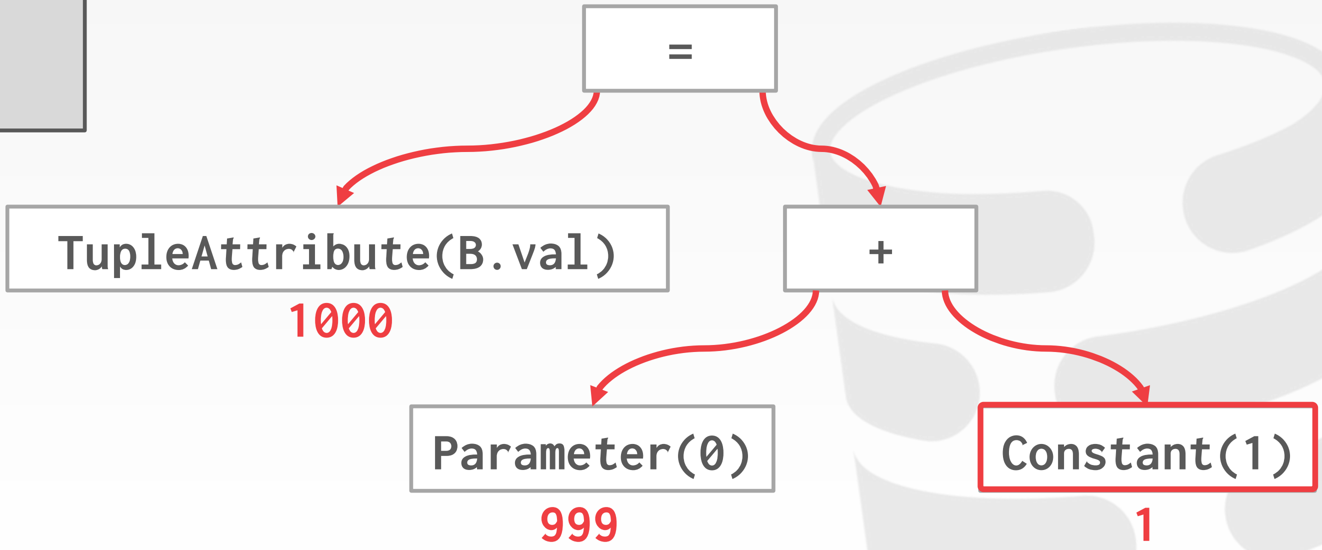


PREDICATE INTERPRETATION

```
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
```

Execution Context

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

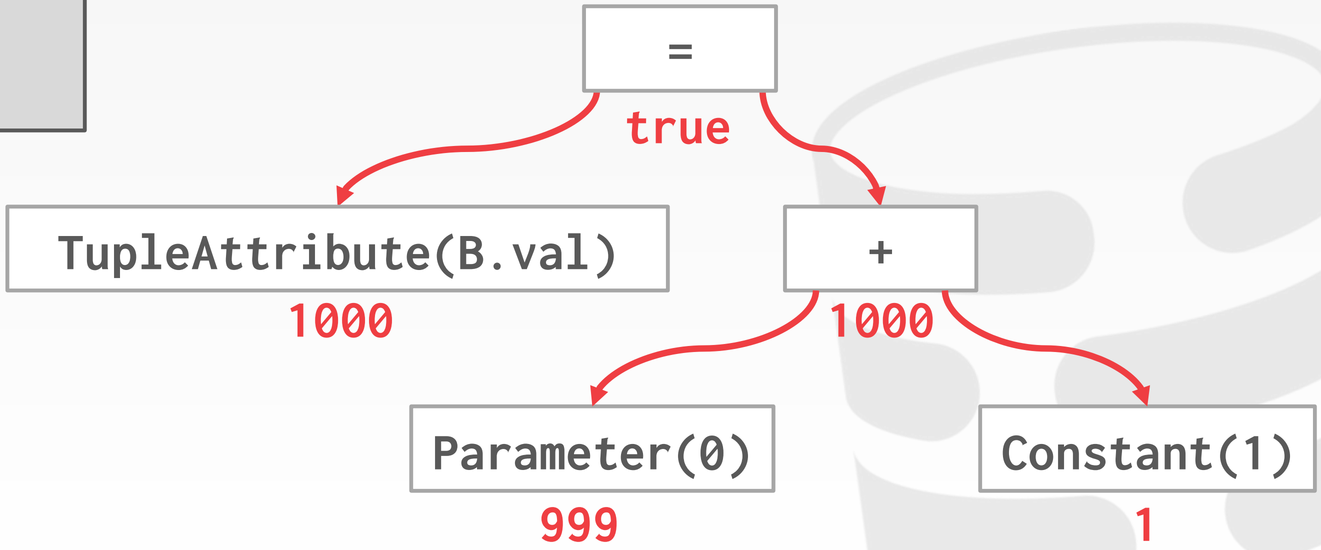


PREDICATE INTERPRETATION

```
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
```

Execution Context

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



CODE SPECIALIZATION

Any CPU intensive entity of database can be natively compiled if they have a similar execution pattern on different inputs.

→ Access Methods
→ Stored Procedures
→ Operator Execution
→ Predicate Evaluation
→ Logging Operations

BENEFITS

Attribute types are known ***a priori***.

→ Data access function calls can be converted to inline pointer casting.

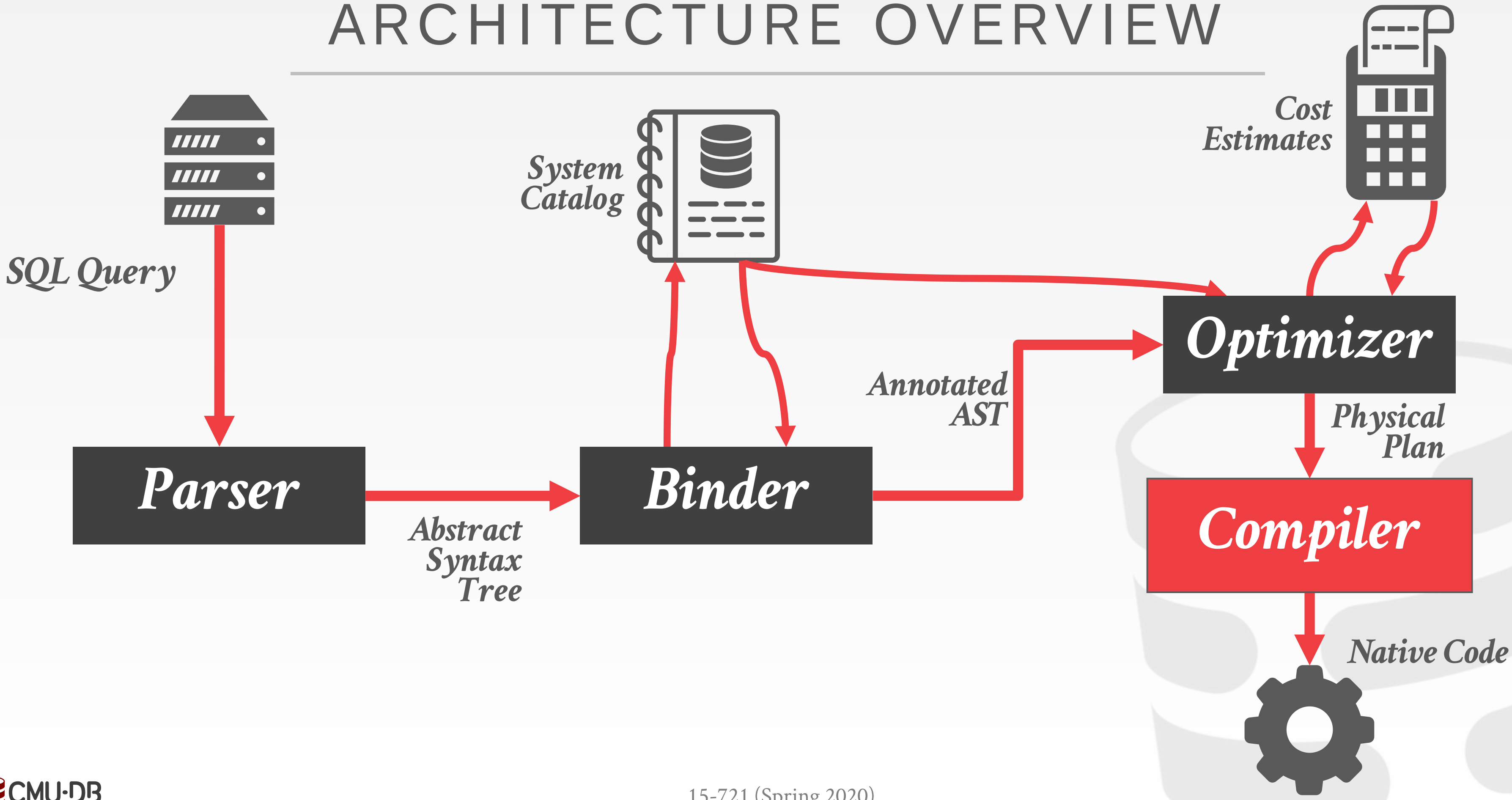
Predicates are known ***a priori***.

→ 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.

ARCHITECTURE OVERVIEW



CODE GENERATION

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 – 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

OPERATOR TEMPLATES

```
SELECT * FROM A WHERE A.val = ? + 1
```

OPERATOR TEMPLATES

Interpreted Plan

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

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.*

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.*

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)
```

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.

→ Concurrency Control
→ Logging / Checkpoints
→ Indexes

EVALUATION

Generic Iterators
→ Canonical model with generic predicate evaluation.

Optimized Iterators
→ Type-specific iterators with inline predicates.

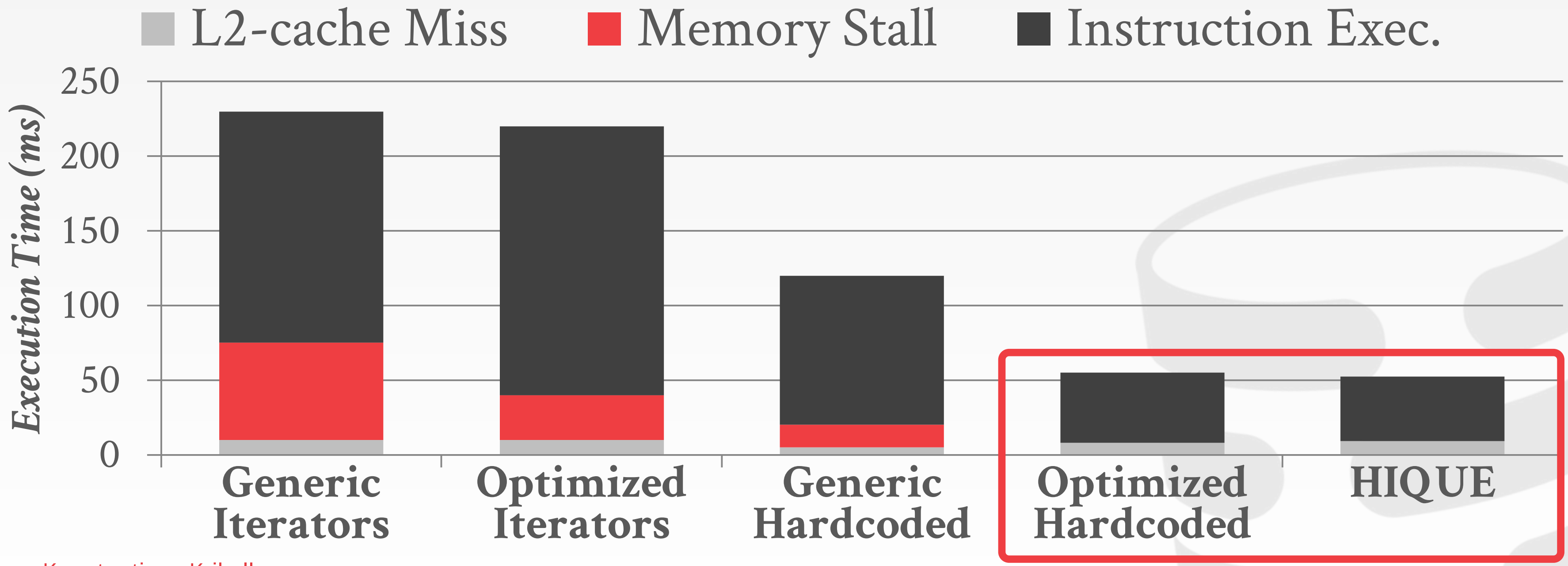
Generic Hardcoded
→ Handwritten code with generic iterators/predicates.

Optimized Hardcoded
→ Direct tuple access with pointer arithmetic.

HIQUE
→ Query-specific specialized code.

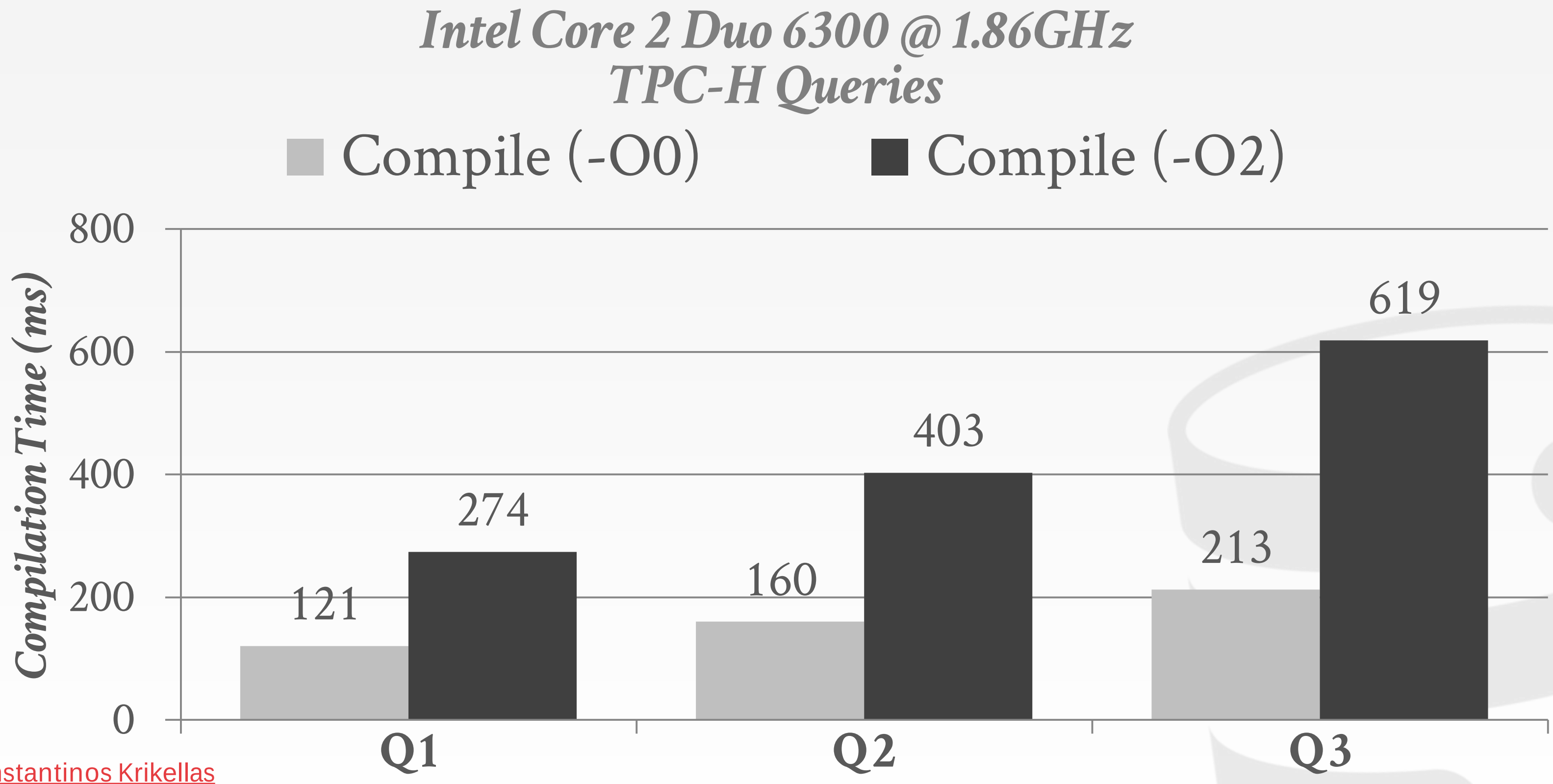
QUERY COMPILATION EVALUATION

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



Source: Konstantinos Krikellas

QUERY COMPILATION COST



Source: Konstantinos Krikellas

OBSERVATION

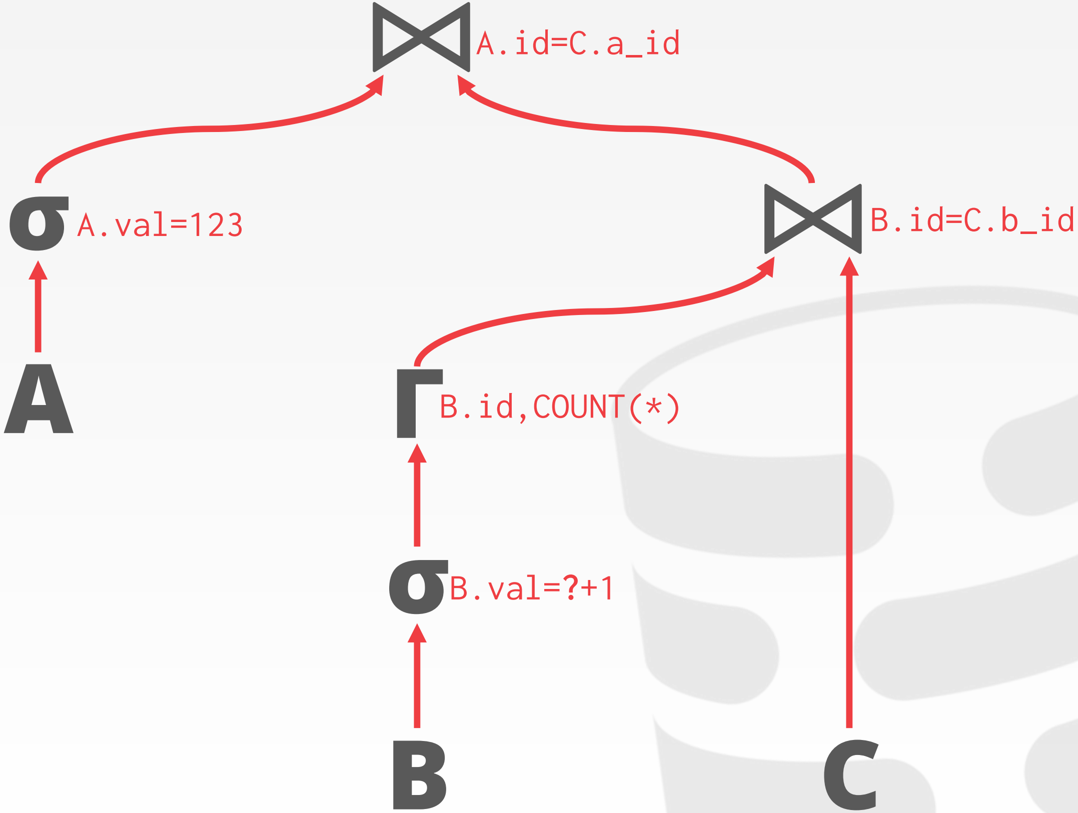
Relational operators are a useful way to reason about a query but are not the most efficient way to execute it.

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

HIQUE does not support for full pipelining…

PIPELINED OPERATORS

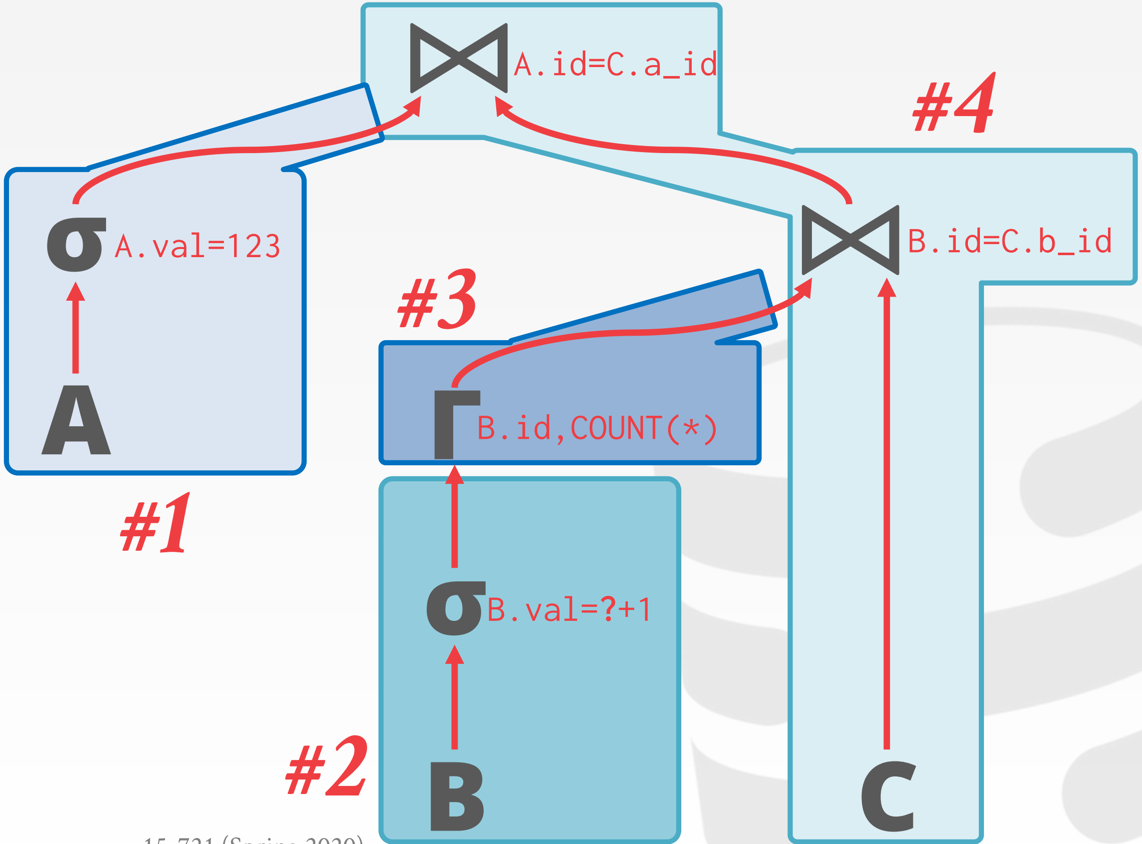
```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```



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



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.
→ Push-based vs. Pull-based
→ Data Centric vs. Operator Centric

EFFICIENTLY COMPILING EFFICIENT QUERY PLANS FOR MODERN HARDWARE
VLDB 2011

LLVM

Collection of modular and reusable compiler and toolchain technologies.

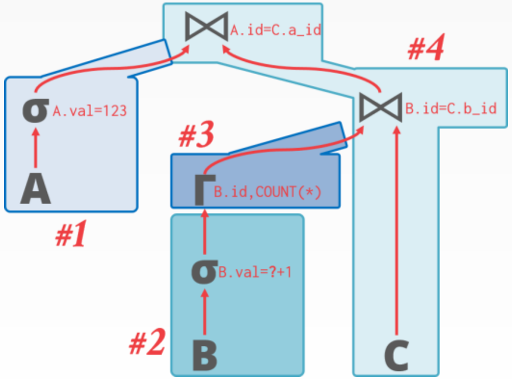
Core component is a low-level programming language (IR) that is like assembly.

Not all the DBMS components need to be written in LLVM IR.

→ LLVM code can make calls to C++ code.

PUSH-BASED EXECUTION

```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
```



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

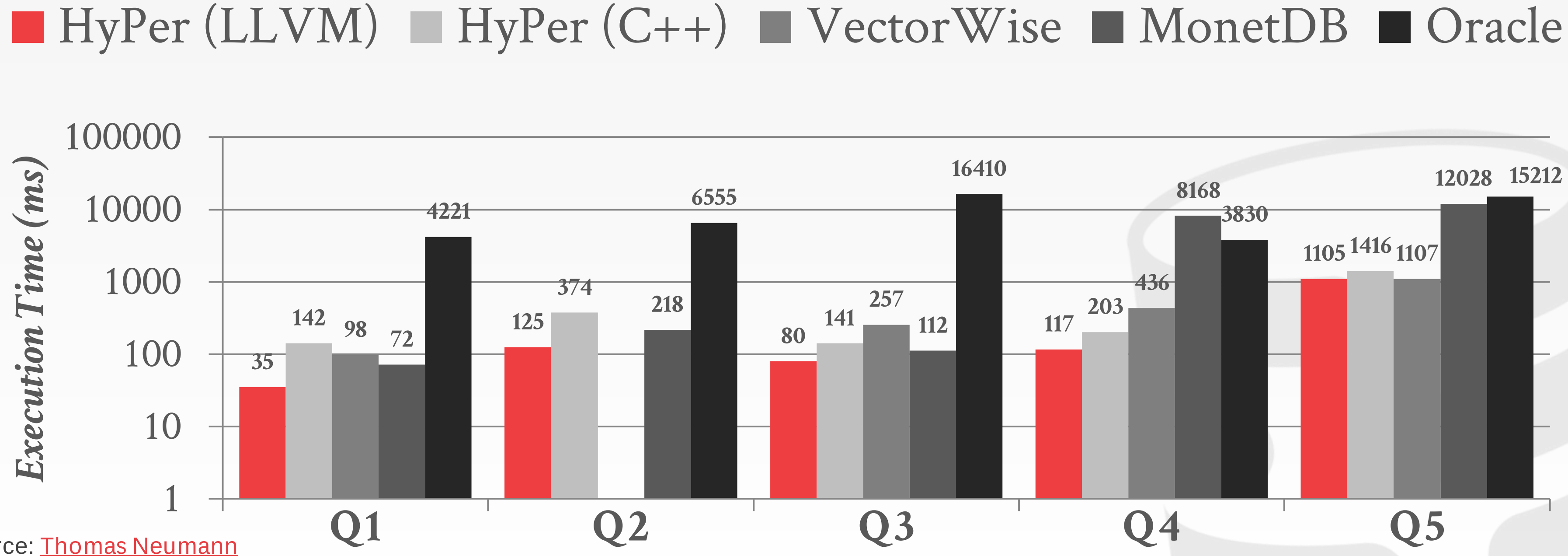
```
for t in Γ(B.id):
  Materialize t in HashTable ⋈(B.id=C.b_id)
```

#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)
```

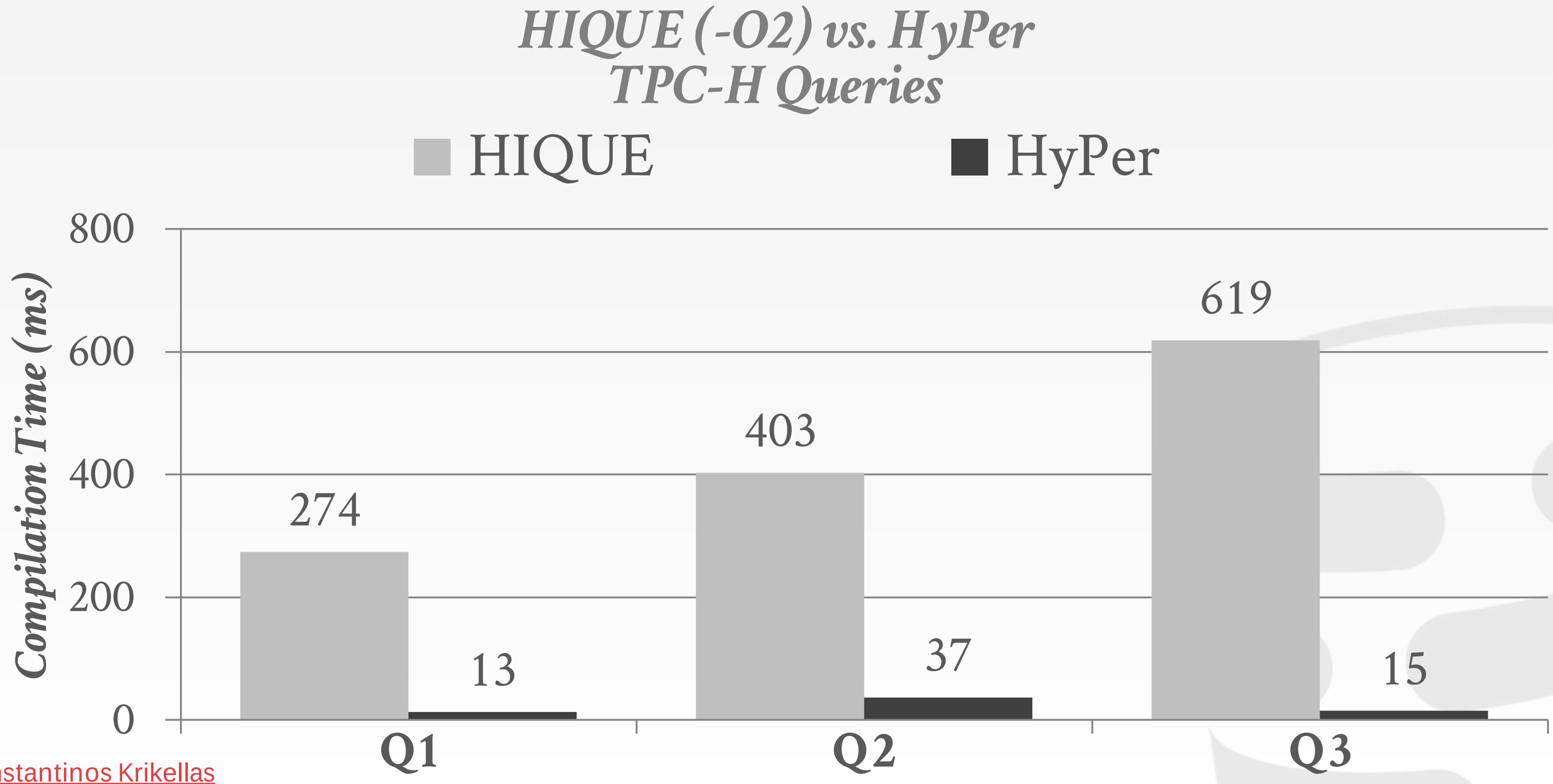
QUERY COMPILATION EVALUATION

Dual Socket Intel Xeon X5770 @ 2.93GHz
TPC-H Queries



Source: Thomas Neumann

QUERY COMPILATION COST



Source: Konstantinos Krikellas

QUERY COMPILATION COST

LLVM's 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

First generate the LLVM IR for the query and then immediately start executing the IR using an interpreter.

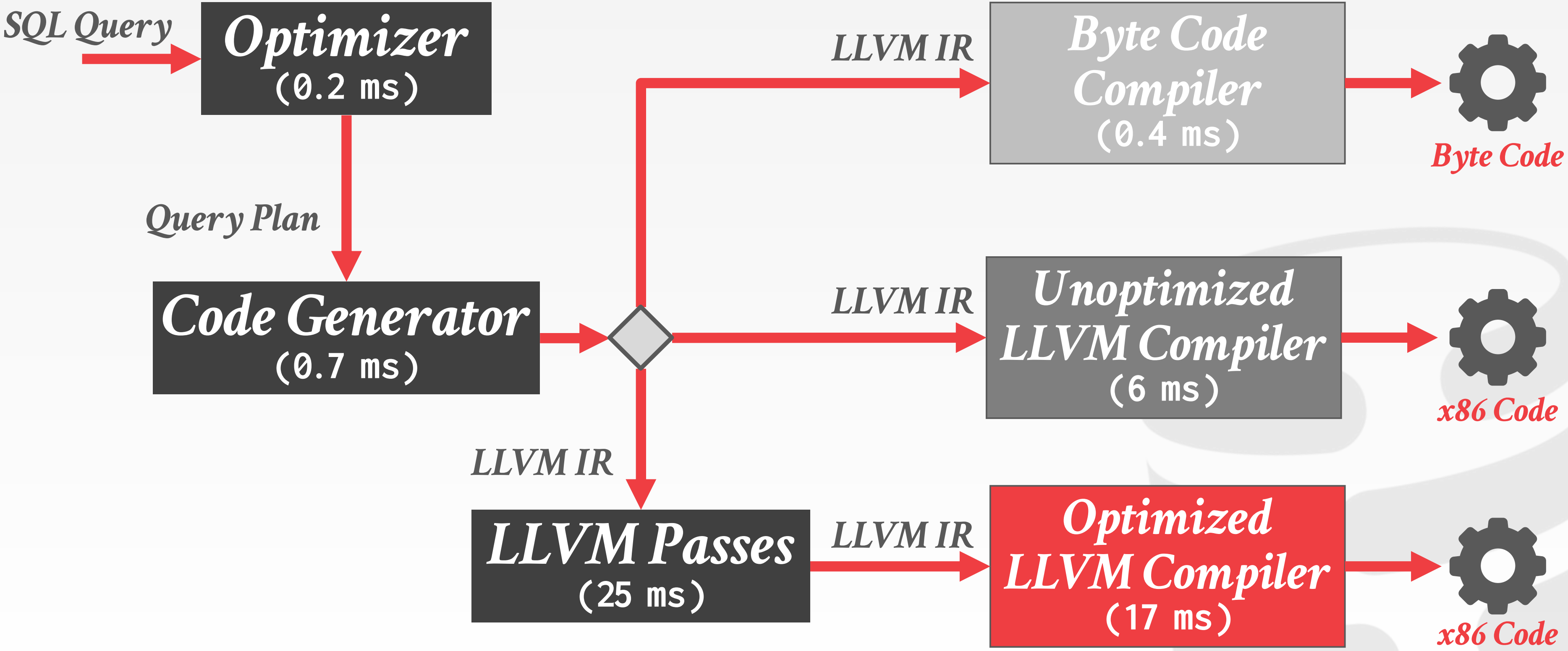
Then the DBMS compiles the query in the background.

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

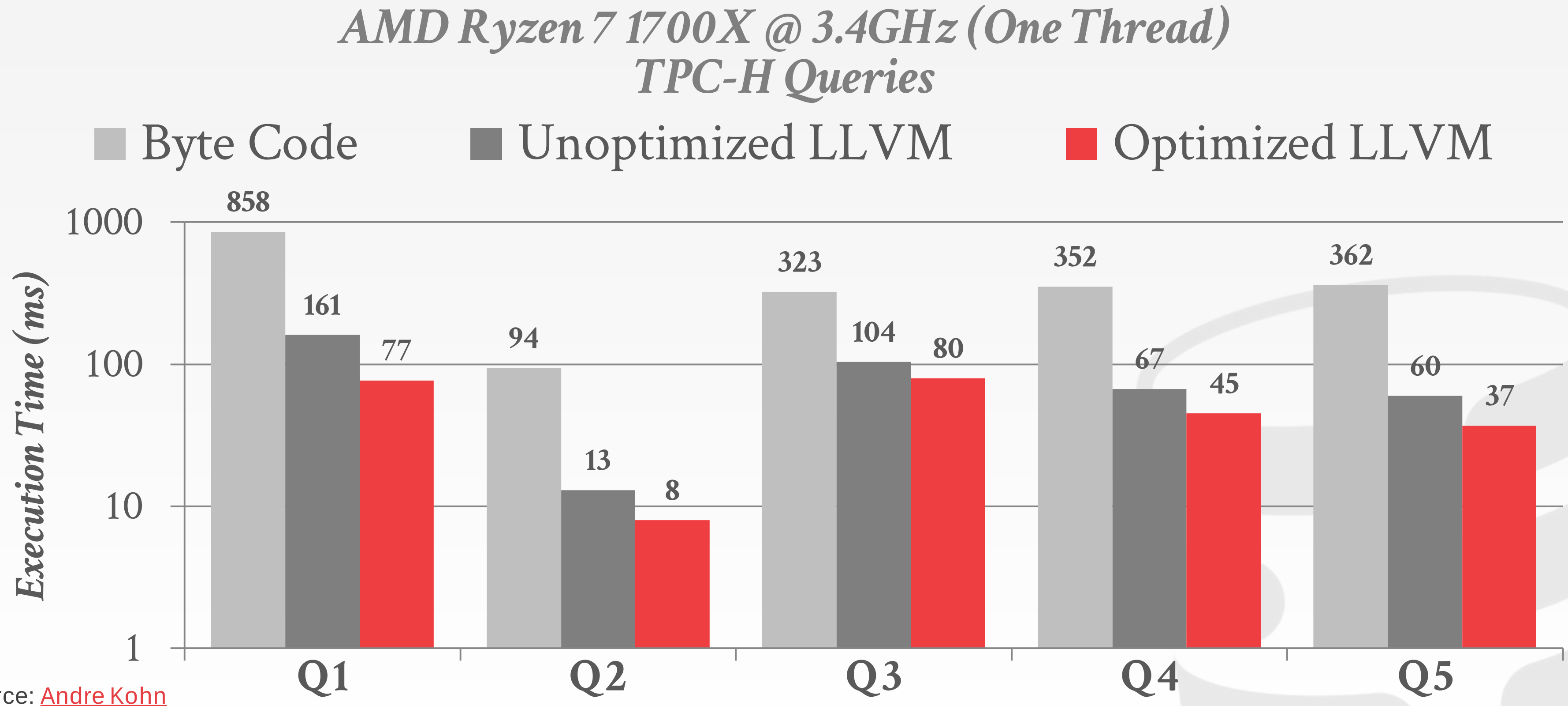
→ For each morsel, check to see whether the compiled version is available.

ADAPTIVE EXECUTION OF COMPILED QUERIES
ICDE 2018

HYPER – ADAPTIVE EXECUTION



HYPER – ADAPTIVE EXECUTION



Source: Andre Kohn

REAL-WORLD IMPLEMENTATIONS

Custom

IBM System R

Oracle

Microsoft Hekaton

Actian Vector

JVM-based

Apache Spark

Neo4j

Splice Machine

Presto

LLVM-based

MemSQL

VitesseDB

PostgreSQL (2018)

Cloudera Impala

Peloton

CMU's DBMS 2.0

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 DB2:

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

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

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

ACTIAN VECTOR

Pre-compiles thousands of "primitives" that perform basic operations on typed data.

→ Example: Generate a vector of tuple ids by applying a less than operator on some column of a particular type.

The DBMS then executes a query plan that invokes these primitives at runtime.

→ Function calls are amortized over multiple tuples

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

ACTIAN VECTOR

```
size_t scan_lessthan_int32(int *res, int32_t *col, int32_t val) {
  size_t k = 0;
  for (size_t i = 0; i < n; i++)
    if (col[i] < val) res[k++] = i;
  return (k);
}
```

```
size_t scan_lessthan_double(int *res, int32_t *col, double val) {
  size_t k = 0;
  for (size_t i = 0; i < n; i++)
    if (col[i] < val) res[k++] = i;
  return (k);
}
```

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.

SPARK SQL: RELATIONAL DATA PROCESSING IN SPARK
SIGMOD 2015

JAVA DATABASES

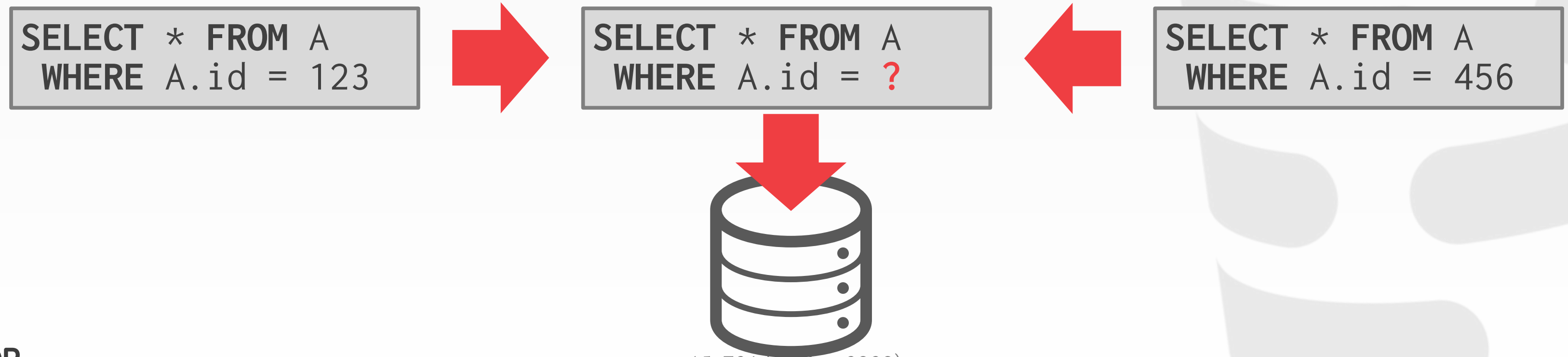
There are several JVM-based DBMSs that contain custom code that emits JVM bytecode directly.

→ Neo4j
→ Splice Machine
→ Presto
→ Derby

MEMSQL (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.



MEMSQL (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.

The DSL then gets converted into a second language of opcodes.
→ MemSQL Bit Code (MBC)
→ Think of this as JVM byte code.

Finally 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

CLOUDERA IMPALA

LLVM JIT compilation for predicate evaluation and record parsing.

→ Not sure if they are also doing operator compilation.

Optimized record parsing is important for Impala because they need to handle multiple data formats stored on HDFS.

IMPALA: A MODERN, OPEN-SOURCE SQL ENGINE FOR HADOOP
CIDR 2015

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

PELOTON (2017)

HyPer-style full compilation of the entire query plan using the LLVM .

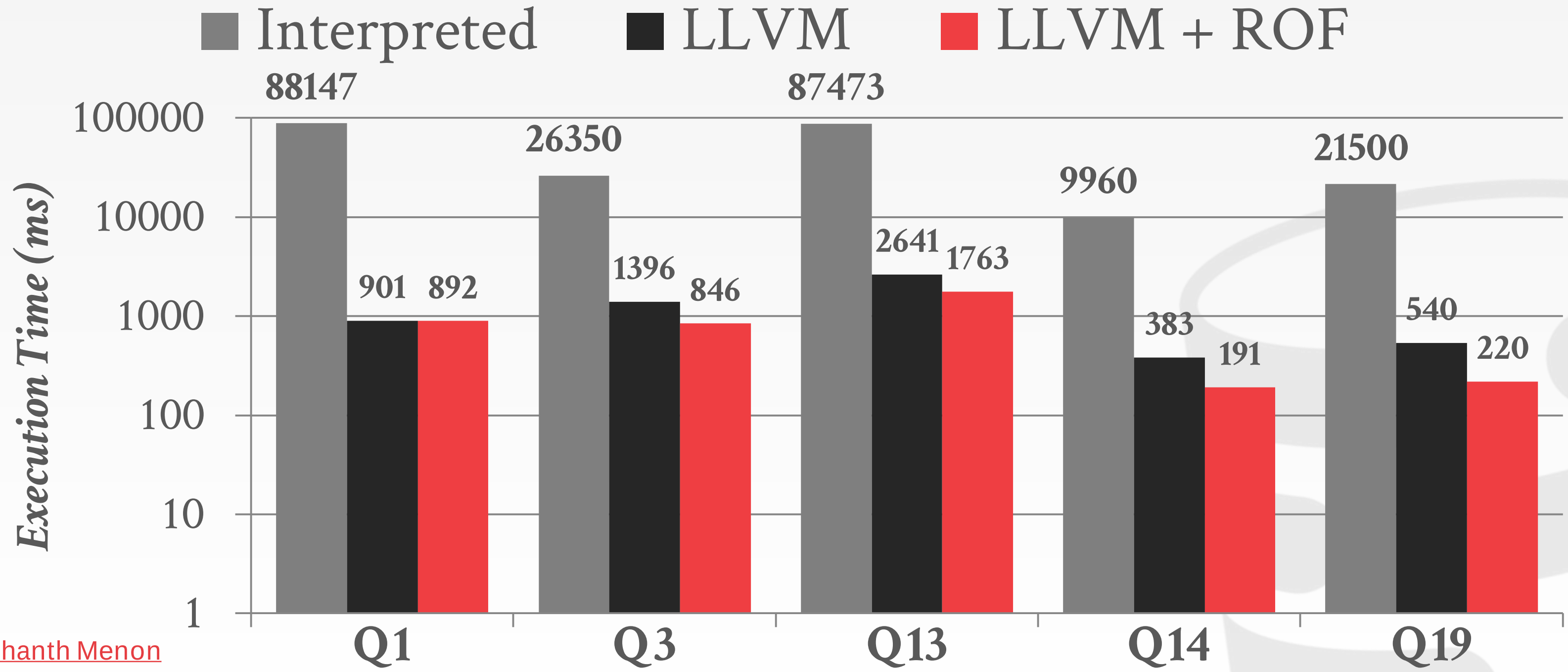
Relax the pipeline breakers create mini-batches for operators that can be vectorized.

Use software pre-fetching to hide memory stalls.

RELAXED OPERATOR FUSION FOR IN-MEMORY DATABASES: MAKING COMPILATION, VECTORIZATION, AND PREFETCHING WORK TOGETHER AT LAST
VLDB 2017

PELOTON (2017)

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



Source: Prashanth Menon

UNNAMED CMU DBMS (2019)

MemSQL-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.

UNNAMED (

```
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

UNNAMED CMU DBMS (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

PARTING THOUGHTS

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

The 2016 version of MemSQL is the best query compilation implementation out there.

Any new DBMS that wants to compete has to implement query compilation.

NEXT CLASS

Vectorization