Lecture #09

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Query Compilation & Code Generation

Andy Pavlo // 15-721 // Spring 2023

ADMINISTRIVIA

Project #1: Sunday February 26th

Project #2: Sunday April 30th

Project #3
→ Proposals: Wednesday March 1st
→ Updates: Monday April 3rd
→ Final Presentations: TBA

LAST CLASS

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

The research literature from 10 years ago 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
Code Generation / Transpilation
JIT Compilation
Real-world Implementations
Project #3

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

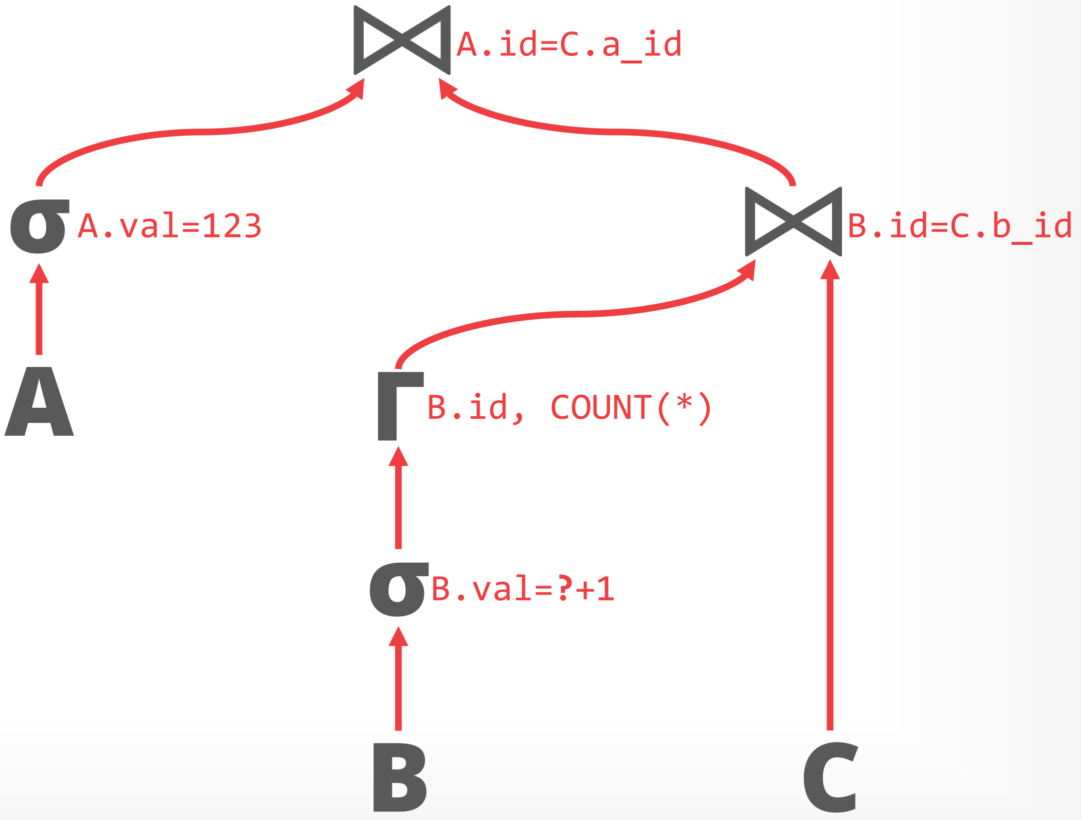
```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT
);
```

```
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)
);
```

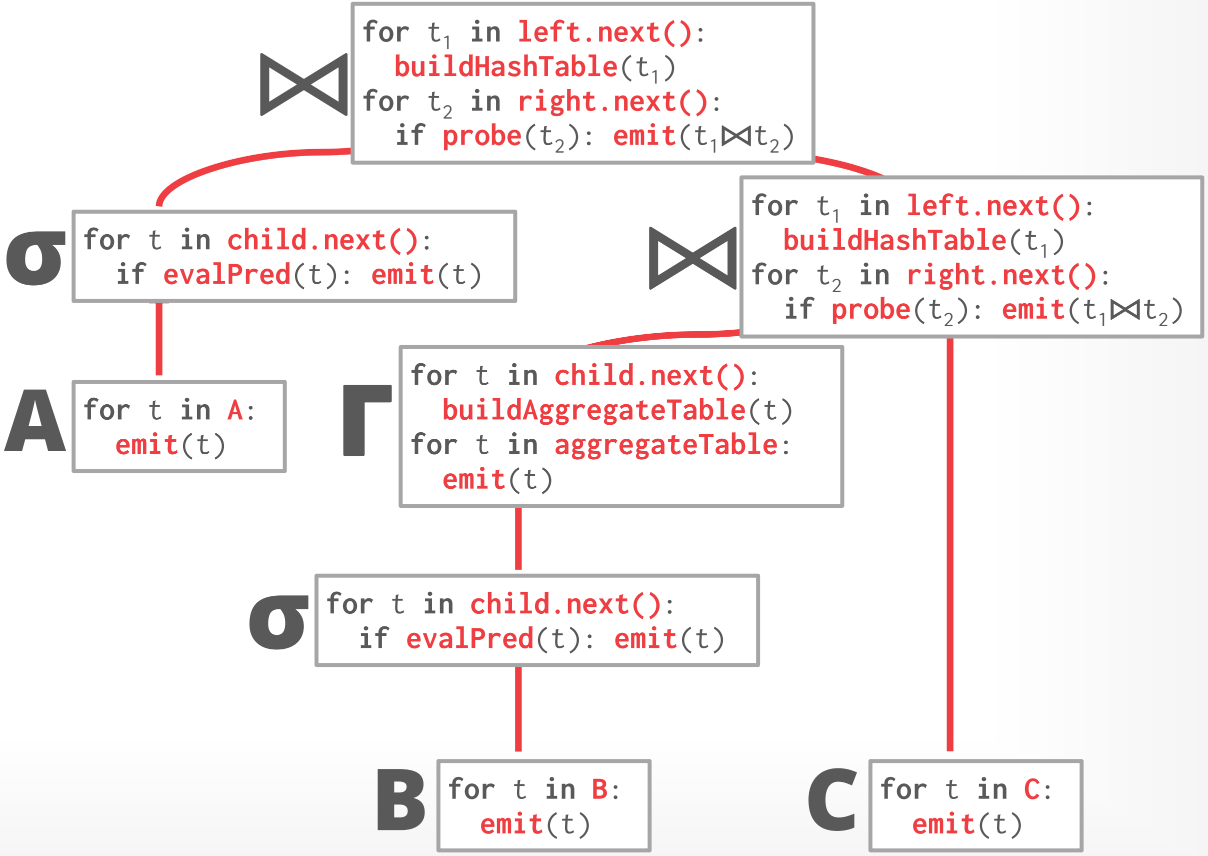
QUERY INTERPRETATION

```
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
```

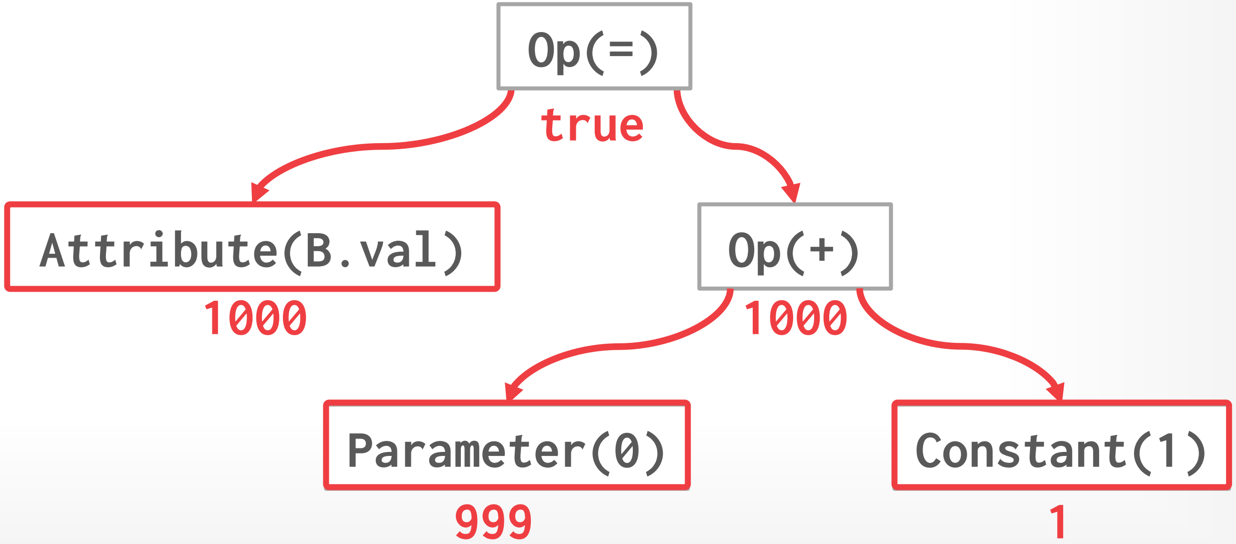


PREDICATE INTERPRETATION

```
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)



CODE SPECIALIZATION

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

```sql
SELECT * FROM A
 WHERE val = 1;
```

```
        Op(=)
       /     \
Attribute(A.val)  Constant(1)
```

```
bool check(val) {
  return (val == 1);
}
```

CODE SPECIALIZATION

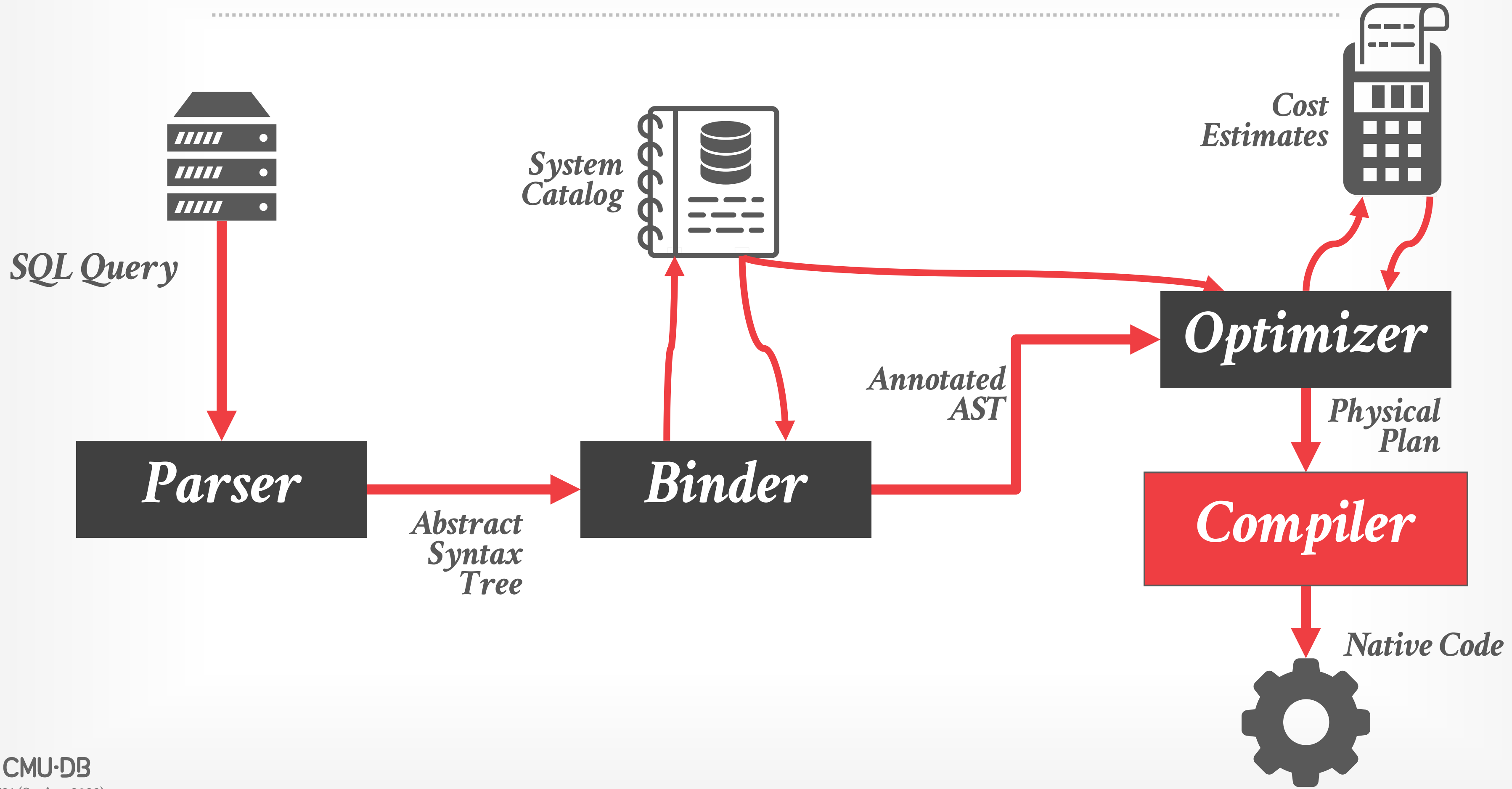
Approach #1: Transpilation

→ 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 .

ARCHITECTURE OVERVIEW



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

HIQUE – OPERATOR TEMPLATES

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

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.

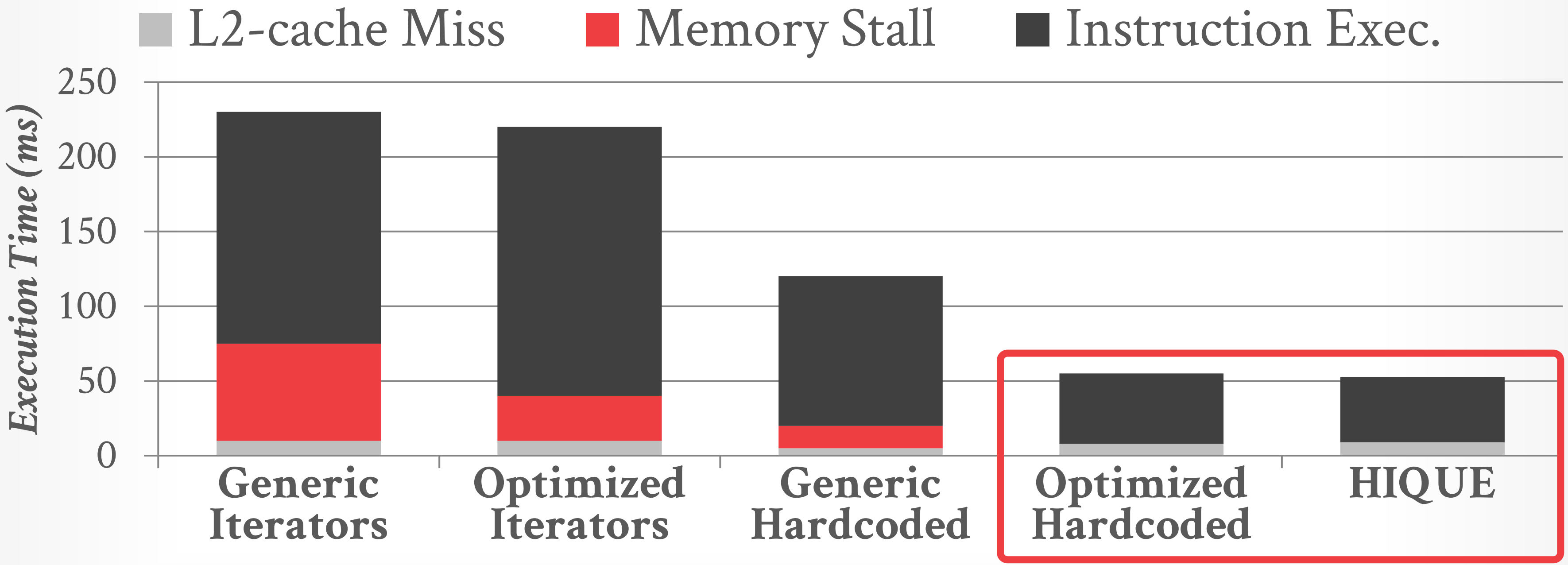
Generic Hardcoded
→ Handwritten code with generic iterators/predicates.

Optimized Hardcoded
→ Direct tuple access with pointer arithmetic.

HIQUE
→ Query-specific specialized code.

QUERY COMPILATION EVALUATION

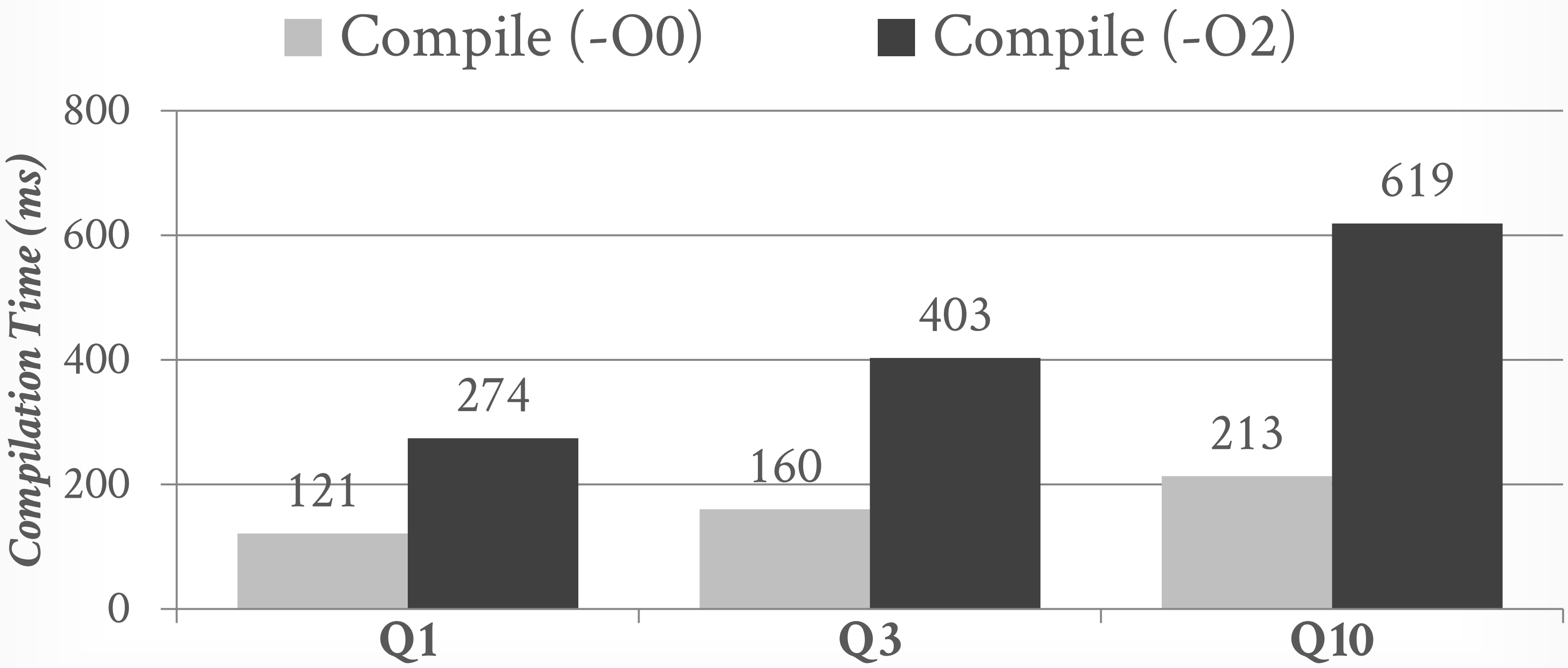
Intel Core 2 Duo 6300 @ 1.86GHz
Join Query: 10k⋈10k→10m



Source: Konstantinos Krikellas

QUERY COMPILATION COST

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



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.

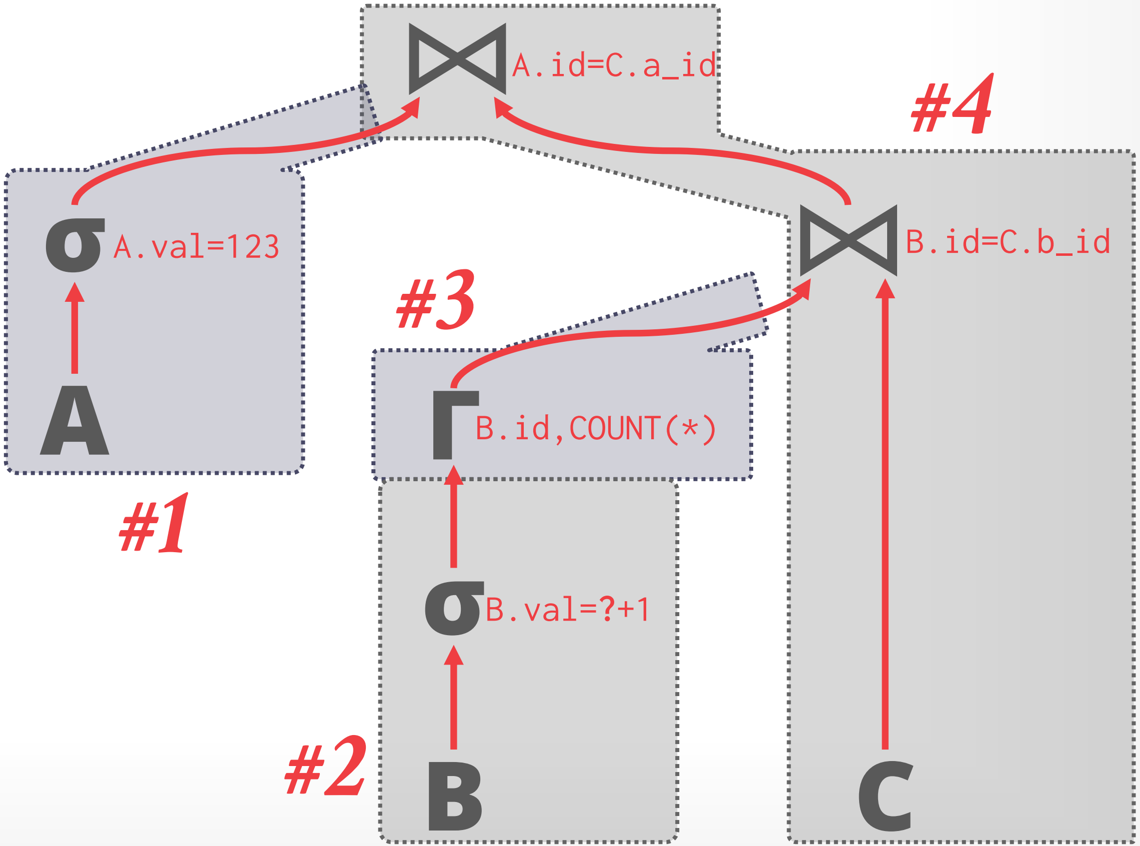
Aggressive operator function within pipelines to keep a tuple in CPU registers for as long as possible.
→ Push-based vs. Pull-based
→ Data Centric vs. Operator Centric

EFFICIENTLY COMPILING EFFICIENT QUERY PLANS FOR MODERN HARDWARE
VLDB 2011

PIPELINED OPERATORS

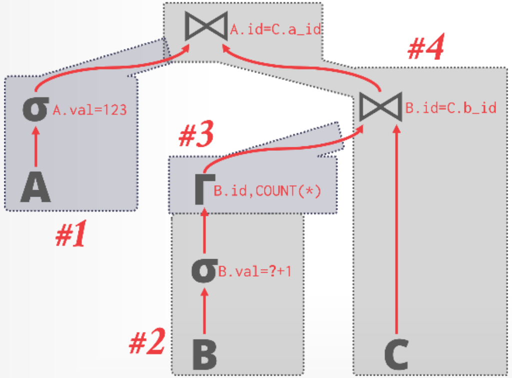
```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Pipeline Boundaries



PUSH-BASED EXECUTION

```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
```

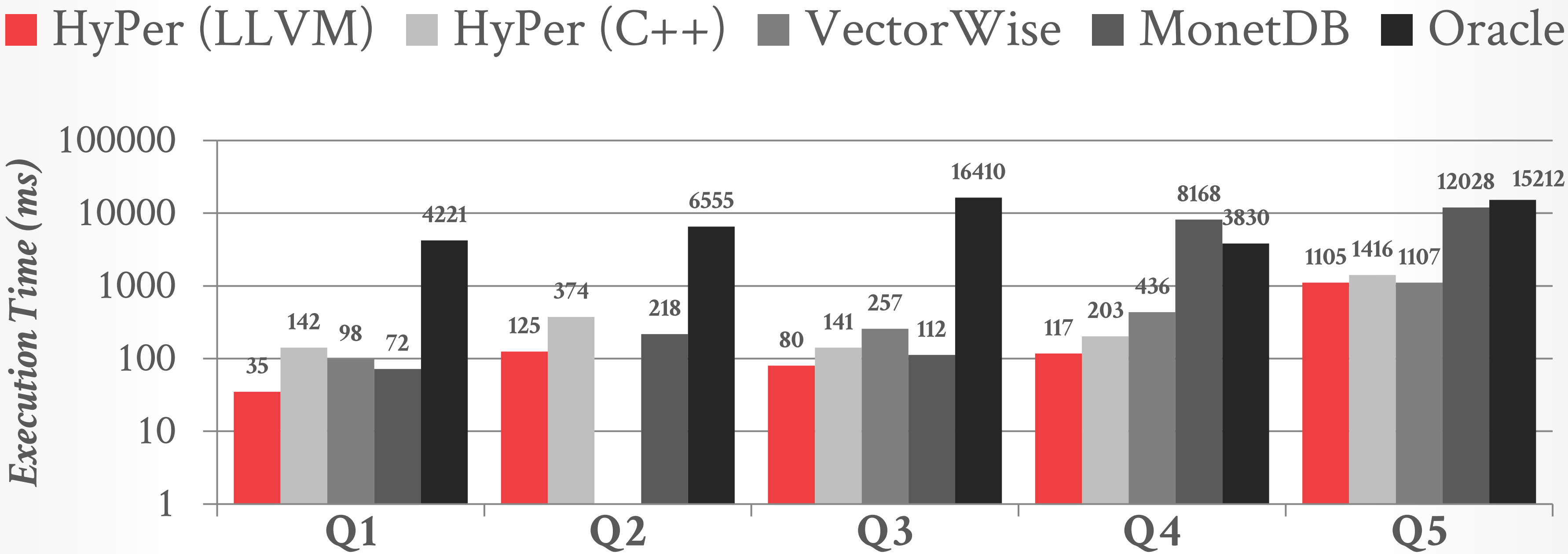


Generated Query Plan

```
#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

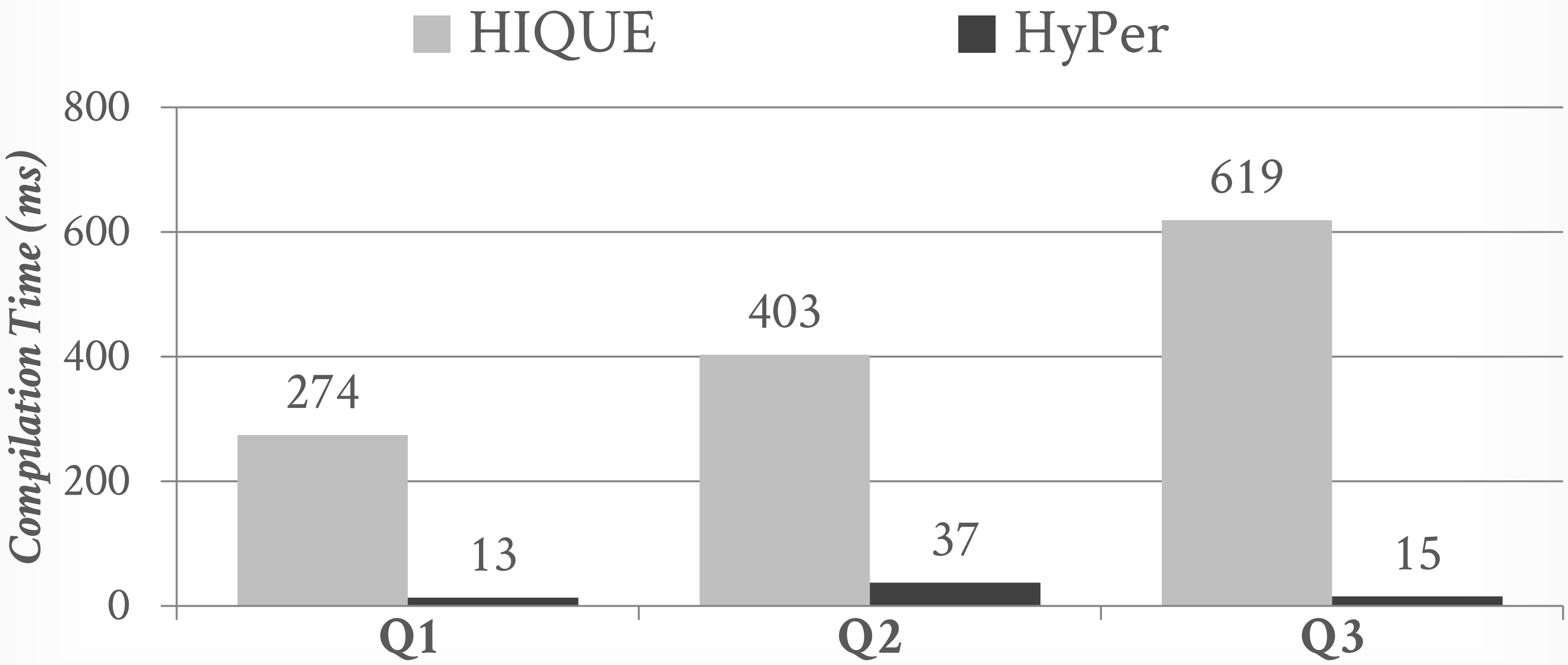
Dual Socket Intel Xeon X5770 @ 2.93GHz
TPC-H Queries (Scalefactor=1)



Source: Thomas Neumann

QUERY COMPILATION COST

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



Source: Konstantinos Krikellas

QUERY COMPILATION COST

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.

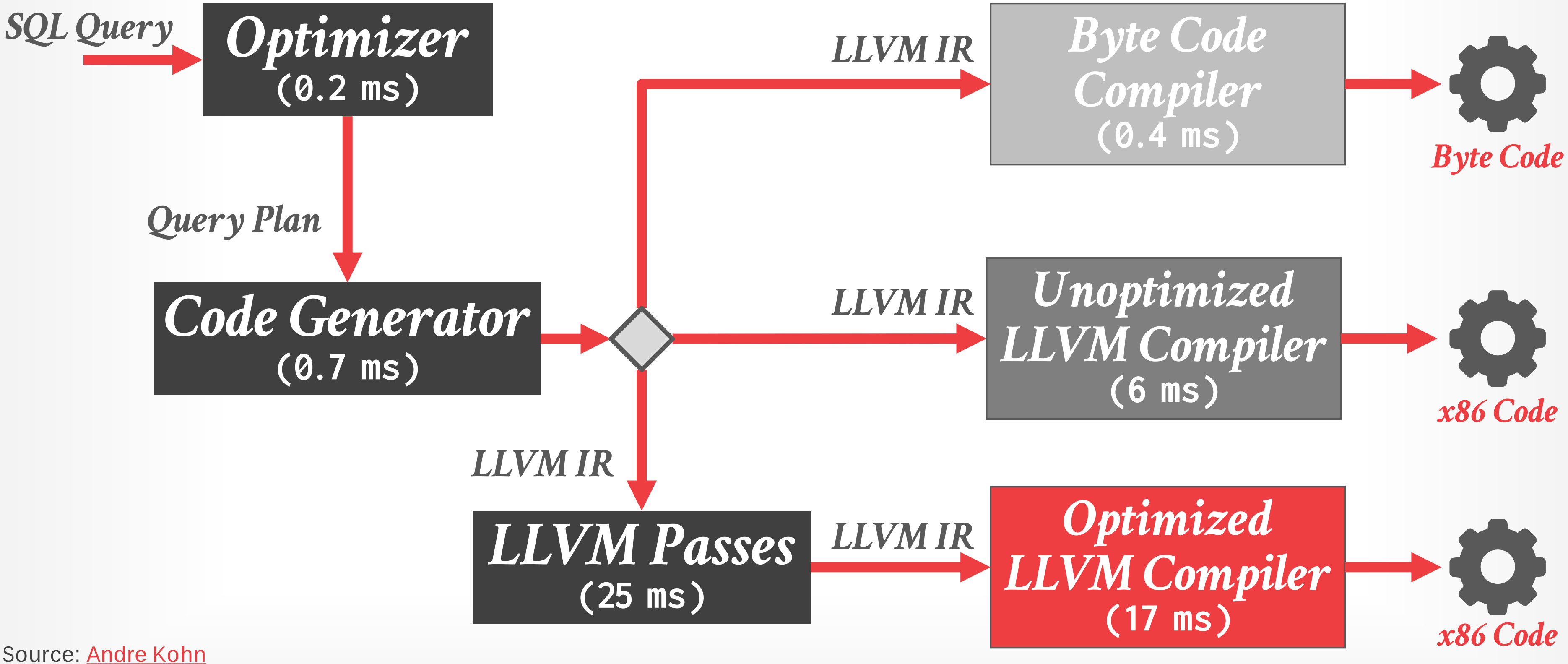
Then the DBMS compiles the query in the background.

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

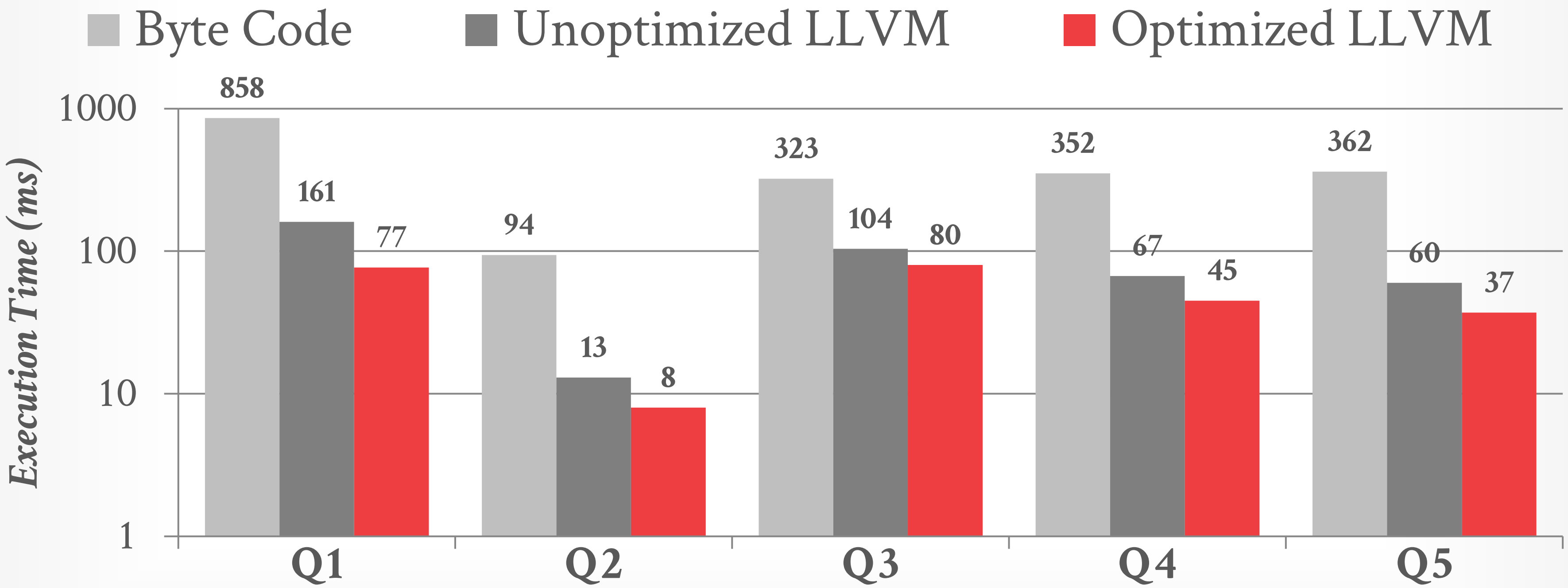
HYPER – ADAPTIVE EXECUTION



Source: Andre Kohn

HYPER – ADAPTIVE EXECUTION

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



Source: Andre Kohn

REAL-WORLD IMPLEMENTATIONS

Custom

IBM System R
Actian Vector
Amazon Redshift
Oracle
Microsoft Hekaton
SQLite
TUM Umbra

JVM-based

Apache Spark
Neo4j
Splice Machine
Presto / Trino

LLVM-based

SingleStore
VitesseDB
PostgreSQL (2018)
CMU Peloton
CMU NoisePage

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 b
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

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

Pre-compiles thousands of "primitives" that perform

```
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);
}
```

The DBMS then executes a query plan that invokes

```
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

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.

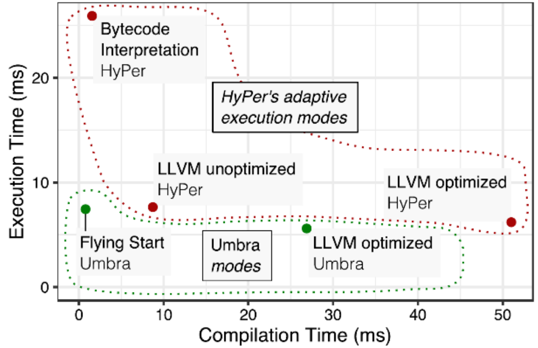


Fig. 14 Umbra's versus HyPer's execution modes. Comparison of time taken for compilation and achieved execution time for Umbra's and HyPer's execution modes on TPC-H query 3. SF = 1, Threads = 20

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

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

JAVA DATABASES

There are several JVM-based DBMSs that contain custom code that emits JVM bytecode directly.
→ Neo4j
→ Splice Machine
→ Presto / Trino
→ Derby

This functionally the same as generating LLVM IR.

SINGLESTORE (PRE-2016)

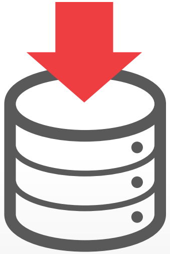
Performs the same C/C++ code generation as HIQUE and then invokes gcc.

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
```



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

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 .

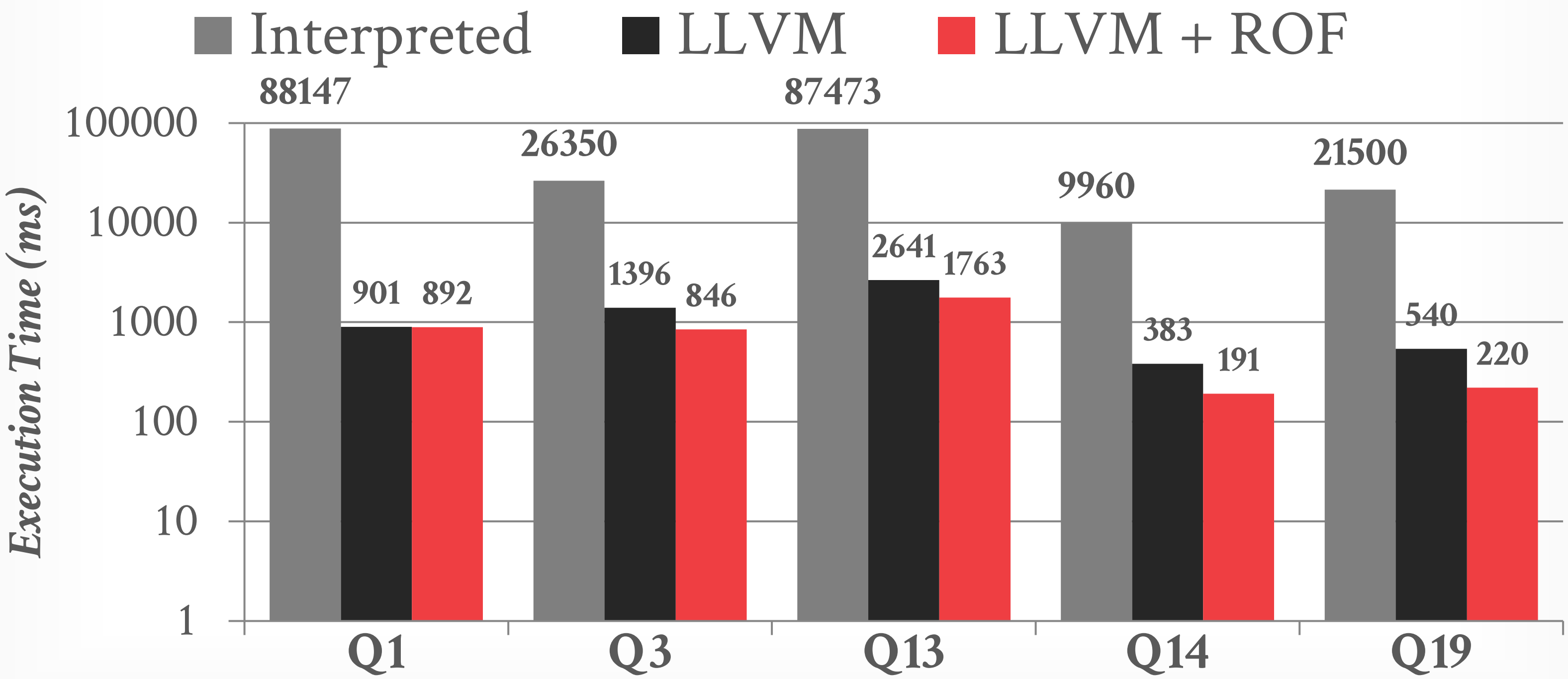
Relax the pipeline breakers create mini-batches for operators that can be vectorized.

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

CMU PELOTON (2017)

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



Source: Prashanth Menon

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
```

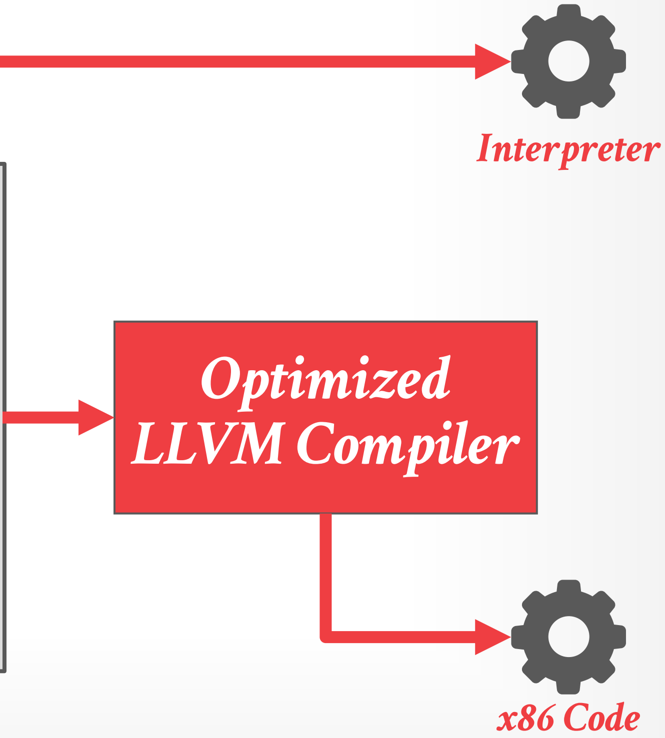
Source: Prashanth Menon

CMU NOISEPAGE (2019)

```sql
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
```



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 vs. Compilation

PROJECT #3

Group project to implement some substantial component or feature in a DBMS.

Projects should incorporate topics discussed in this course as well as from your own interests.

Each group must pick a project that is unique from their classmates.

PROJECT #3 – DELIVERABLES

Proposal

Status Update

Design Document

Final Presentation

PROJECT #3 – PROPOSAL

Five-minute presentation to the class that discusses the high-level topic.

Each proposal must discuss:
→ High-level overview and system archicture of your project.
→ How you will test whether your implementation is correct.
→ What workloads you will use for your project.

PROJECT #3 – STATUS UPDATE

Five-minute presentation to update the class about the current status of your project.

Each presentation should include:

→ Current development status.
→ Whether your plan has changed and why.
→ Anything that surprised you during coding.

PROJECT #3 – DESIGN DOCUMENT

As part of the status update, you must provide a design document that describes your project implementation:

→ Architectural Design
→ Design Rationale
→ Testing Plan
→ Trade-offs and Potential Problems
→ Future Work

PROJECT #3 – FINAL PRESENTATION

10-minute presentation on the final status of your project during the scheduled final exam.

You should include any performance measurements or benchmarking numbers for your implementation.

Demos are always hot too…

PROJECT TOPICS

Fast Fixed-Point Decimals

Proxy Kernel Bypass

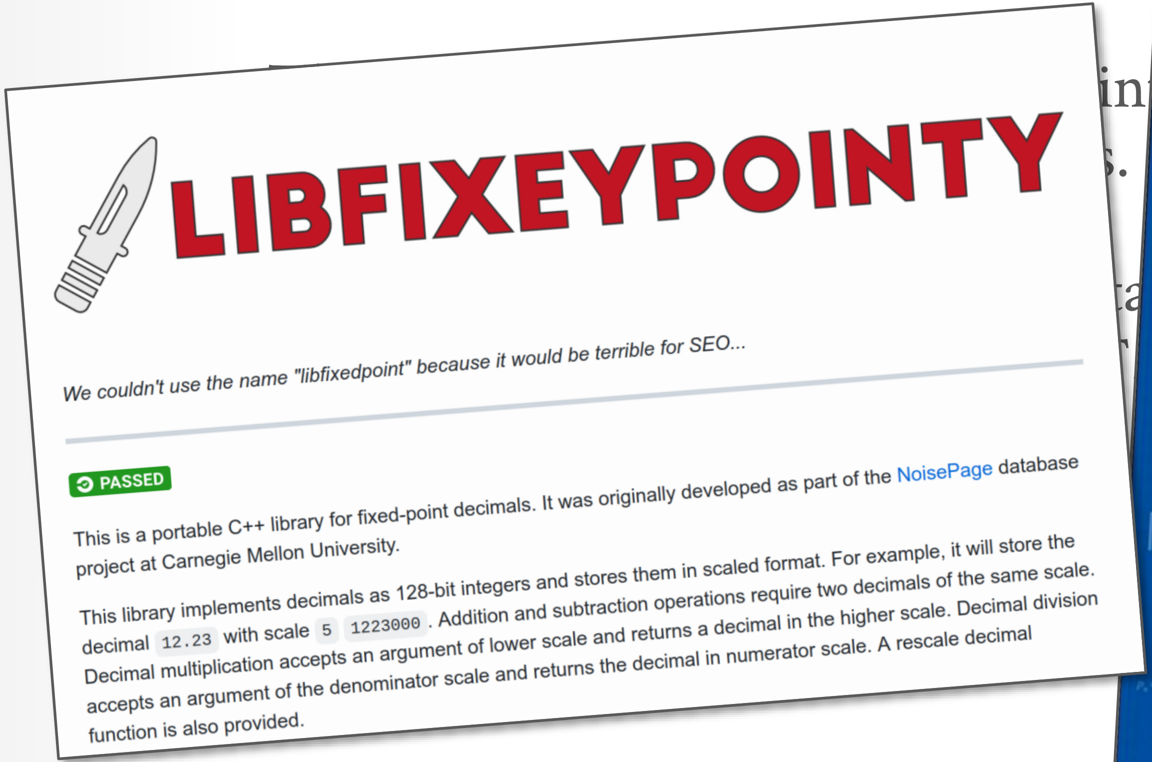
Adaptive Query Opt.

FAST FIXED-POINT DECIMALS

The Germans claim that fixed-point decimals are faster than floating point decimals.

Project: Complete our implementation and integrate into PostgreSQL as UDT.

FAST FIXED-POINT D



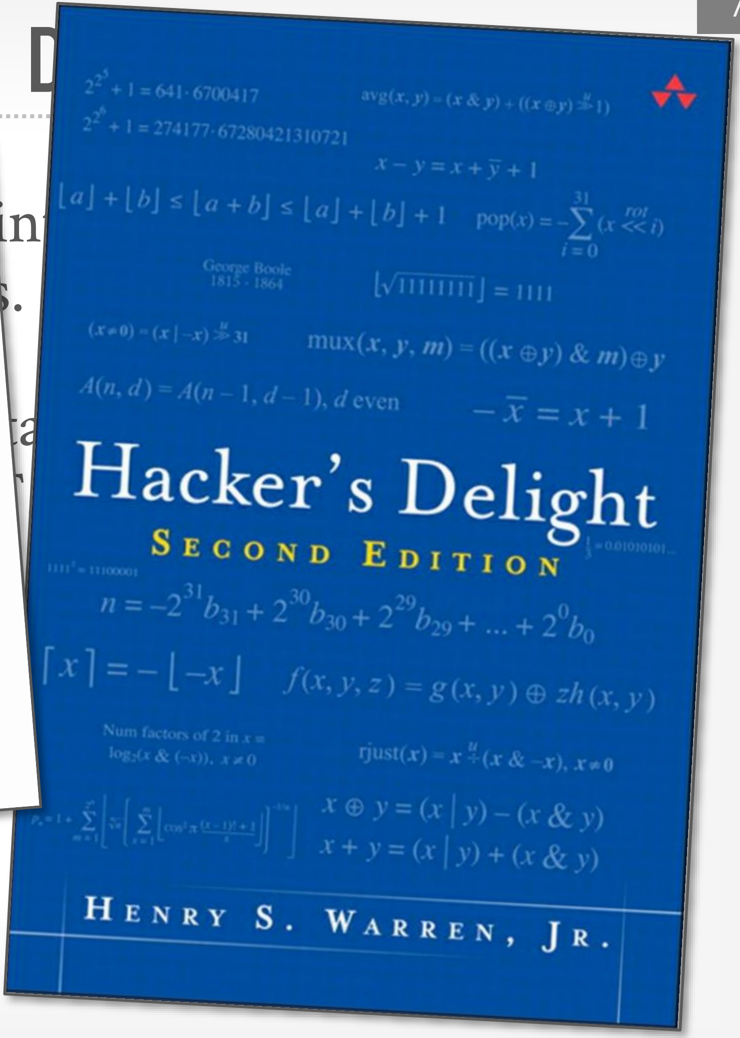
LIBFIXEYPOINTY

We couldn't use the name "libfixedpoint" because it would be terrible for SEO...

PASSED

This is a portable C++ library for fixed-point decimals. It was originally developed as part of the NoisePage database project at Carnegie Mellon University.

This library implements decimals as 128-bit integers and stores them in scaled format. For example, it will store the decimal `12.23` with scale `5` `1223000`. Addition and subtraction operations require two decimals of the same scale. Decimal multiplication accepts an argument of lower scale and returns a decimal in the higher scale. Decimal division accepts an argument of the denominator scale and returns the decimal in numerator scale. A rescale decimal function is also provided.



PROXY KERNEL-BYPASS

We have been working on optimizations for PostgreSQL proxies.

Project: Modify PgBouncer to use io_uring.

→ Matt has existing benchmark scripts to compare against his proxy and Odyssey.

ADAPTIVE QUERY OPTIMIZATION

We want to be able to change a query plan during execution without stopping the query.

Project: Create a PostgreSQL extension that swaps a plan node in the tree with a "dummy" node.
- → New node can either halt execution or generate fake data.
- → An easier approach might be to wrap nodes with "control" nodes that determine whether to call inner node.

HOW TO START

Form a team.

Meet with your team and discuss potential topics.

Look over source code and determine what you will need to implement.

I am able during Spring Break for additional discussion and clarification of the project idea.