Query Compilation & Code Generation

@Andy_Pavlo // 15-721 // Spring 2020
Project #2 Checkpoint: Sunday March 8<sup>th</sup>

Project #2 Final: Sunday March 15<sup>th</sup>

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 $10\times$ faster, the DBMS must execute $90\%$ fewer instructions…

→ To go $100\times$ faster, the DBMS must execute $99\%$ fewer instructions…
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...
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)  
);
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
FROM B
WHERE B.val = ? + 1
GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
FROM B
WHERE B.val = ? + 1
GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id
SELECT * 
FROM A, C, 
(SELECT B.id, COUNT(*) 
FROM B 
WHERE B.val = ? + 1 
GROUP BY B.id) AS B 
WHERE A.val = 123 
AND A.id = C.a_id 
AND B.id = C.b_id

**Execution Context**

<table>
<thead>
<tr>
<th>Current Tuple</th>
<th>Query Parameters</th>
<th>Table Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>(123, 1000)</td>
<td>(int:999)</td>
<td>B&gt;(int:id, int:val)</td>
</tr>
</tbody>
</table>

**TupleAttribute(B.val)**

- **=**
- **+**
- **Parameter(0)**
- **Constant(1)**
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
FROM B
WHERE B.val = ? + 1
GROUP BY B.id) AS B
WHERE A.val = 123
AND A.id = C.a_id
AND B.id = C.b_id

Execution Context

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

TupleAttribute(B.val) = 1000
+ Parameter(0)
Constant(1)
**Execution Context**

```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
```

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

**Precedence and Evaluation**

- `Parameter(0) + Constant(1) = 999`
- `TupleAttribute(B.val) = 1000`
- `A.val = 123`
- `A.id = C.a_id`
- `B.id = C.b_id`
**Execution Context**

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

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:999)

Table Schema
B→(int:id, int:val)

TupleAttribute(B.val)

= 1000

true

Parameter(0)

+ 999

Constant(1)

1000
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.
**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.
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.
SELECT * FROM A WHERE A.val = ? + 1
Interpreted Plan

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

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

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

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

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

### Templated Plan

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

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

1. Traverse predicate tree and pull values up.
2. If tuple value, calculate the offset of the target attribute.
3. Perform casting as needed for comparison operators.
4. Return true / false.
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 $\Join$ 10k $\rightarrow$ 10m

Source: Konstantinos Krikellas
**QUERY COMPILATION COST**

*Intel Core 2 Duo 6300 @ 1.86GHz*  
*TPC-H Queries*

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

Source: Konstantinos Krikellas

<table>
<thead>
<tr>
<th></th>
<th>Compile (-O0)</th>
<th>Compile (-O2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>121</td>
<td>274</td>
</tr>
<tr>
<td>Q2</td>
<td>160</td>
<td>403</td>
</tr>
<tr>
<td>Q3</td>
<td>213</td>
<td>619</td>
</tr>
</tbody>
</table>
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...
SELECT *
FROM A, C,
(SELECT B.id, COUNT(*)
    FROM B
    WHERE B.val = ? + 1
    GROUP BY B.id) AS B
WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
SELECT * 
FROM A, C, 
(SELECT B.id, COUNT(*) 
FROM B 
WHERE B.val = ? + 1 
GROUP BY B.id) AS B 
WHERE A.val = 123 
AND A.id = C.a_id 
AND B.id = C.b_id
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
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

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 \( \land (A.id=C.a_id) \)

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

#3
for t in \( \Gamma(B.id) \):
  Materialize t in HashTable \( \land (B.id=C.b_id) \)

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

15-721 (Spring 2020)
QUERY COMPILATION EVALUATION

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

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

Source: Thomas Neumann
QUERY COMPILATION COST

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

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.
HYPER – ADAPTIVE EXECUTION

SQL Query → **Optimizer** (0.2 ms) → Query Plan

**Code Generator** (0.7 ms) → LLVM IR

LLVM IR → **Byte Code Compiler** (0.4 ms) → Byte Code

LLVM IR → **Unoptimized LLVM Compiler** (6 ms) → x86 Code

LLVM IR → **LLVM Passes** (25 ms) → LLVM IR

LLVM IR → **Optimized LLVM Compiler** (17 ms) → x86 Code
HYPER – ADAPTIVE EXECUTION

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

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

Source: Andre Kohn
# Real-World Implementations

<table>
<thead>
<tr>
<th>Custom</th>
<th>JVM-based</th>
<th>LLVM-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM System R</td>
<td>Apache Spark</td>
<td>MemSQL</td>
</tr>
<tr>
<td>Oracle</td>
<td>Neo4j</td>
<td>VitesseDB</td>
</tr>
<tr>
<td>Actian Vector</td>
<td>Presto</td>
<td>Cloudera Impala</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Peloton</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CMU's DBMS 2.0</td>
</tr>
</tbody>
</table>
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
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.
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.
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.

```c
size_t scan_less_than_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);
}
```

```c
size_t scan_less_than_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);
}
```
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.
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
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.
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.
PELOTON (2017)

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

- Interpreted
- LLVM
- LLVM + ROF

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 CMU DBMS (2019)**

### SQL Statement

```sql
SELECT * FROM foo
WHERE colA >= 50
AND colB < 100000;
```

### Compiled Code

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

Source: Prashanth Menon
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
}

Interpreted

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.
Vectorization