15-721 DATABASE SYSTEMS

Lecture #19 – Query Compilation

TODAY’S AGENDA

Background
Code Generation
JIT Compilation (LLVM)
Real World Implementations
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...
The only way that we can achieve such a reduction in the number of instructions is through **code specialization**.

This means generating code that is specific to a particular task in the DBMS.

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
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

**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
```

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

```
equals

TupleAttribute(val)
```

```
+  
Parameter(0)  
Constant(1)
```
PREDICATE INTERPRETATION

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(val)
equals
+ 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)

equals

TupleAttribute(val)

Parameter(0)

Constant(1)

1000

+
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)

**Predicate Interpretation**

```
TupleAttribute(val)
```

**equals**

```
Parameter(0)
```

```
Constant(1)
```

```
+ 
```

```
1000
```
### 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)

### Predicate Interpretation

- **TupleAttribute(val)**
  - **equals**
  - **Parameter(0)** + **Constant(1)**
  - **equals**
    - **1000**
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(val)
1000
equals
Parameter(0)
Constant(1)
999

**Predicate Interpretation**

**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
```

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

**Diagram**:

```
TupleAttribute(val)  
1000  
+  
999  
Parameter(0)  
equals  
Constant(1)
```
**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
```

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

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

Predicates are known \textit{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.
QUERY WORKFLOW

1. **Parser**
   - SQL Query
   - Abstract Syntax Tree

2. **Planner**
   - Logical Plan

3. **Optimizer**
   - Physical Plan

4. **Compiler**
   - Native Code
   - Cost Estimates
QUERY COMPILATION

Choice #1: Code Generation
→ Write code that converts a relational query plan into C/C++ and then run it through a conventional compiler to generate native code.

Choice #2: JIT Compilation
→ Generate an intermediate representation (IR) of the query that can be quickly compiled 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.
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)
```
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.
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.
**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) + 1
    if (val == parameter_value):
        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.
**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) + 1
    if (val == parameter_value):
        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.
**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**

```
tuple_size = ###
predicate_offset = ###
predicate_value = ###
```

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

for t in range(table.num_tuples):
    tuple = table.data + t * tuple_size
    val = (tuple + predicate_offset) + 1
    if (val == parameter_value):
        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.
WHY NOT C++ TEMPLATES?

It is possible to specialize different DBMS components in the system using C++ templates.

Templates are expanded at compile time. The DBMS’s code would have to account for all possible combinations of value types.
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 \( \bowtie \) 10k \( \rightarrow \) 10m

- L2-cache Miss
- Memory Stall
- Instruction Exec.

Execution Time (ms)

- Generic Iterators
- Optimized Iterators
- Generic Hardcoded
- Optimized Hardcoded
- HIQUE

Source: Konstantinos Krikellas
QUERY COMPILATION EVALUATION

Intel Core 2 Duo 6300 @ 1.86GHz
Join Query: 10k\times10k\rightarrow10m

- L2-cache Miss
- Memory Stall
- Instruction Exec.

Source: Konstantinos Krikellas
QUERY COMPILATION COST

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

Compile (-O0)  Compile (-O2)

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 allow 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
LLVM

Collection of modular and reusable compiler and toolchain technologies.

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

Not all of 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

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

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

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

for t3 in C:
  for t2 in \(\bowtie\)(B.id=C.b_id):
    for t1 in \(\bowtie\)(A.id=C.a_id):
      emit(t1Xt2Xt3)
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 $\Join (A.id=C.a_id)$

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

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

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

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

HyPer (LLVM)  HyPer (C++)  VectorWise  MonetDB  ???

Source: Thomas Neumann
QUERY COMPILATION COST

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

<table>
<thead>
<tr>
<th></th>
<th>HIQUE</th>
<th>HyPer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query #1</td>
<td>274</td>
<td>13</td>
</tr>
<tr>
<td>Query #2</td>
<td>403</td>
<td>37</td>
</tr>
<tr>
<td>Query #3</td>
<td>619</td>
<td>15</td>
</tr>
</tbody>
</table>

Source: Konstantinos Krikellas
PRASHANTH’S MICROBENCHMARK

Database: 10 million Tuples
Single-threaded Execution

<table>
<thead>
<tr>
<th></th>
<th>Execution</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong></td>
<td>430</td>
<td>37.6</td>
</tr>
<tr>
<td></td>
<td>2.66</td>
<td></td>
</tr>
<tr>
<td><strong>GROUP-BY</strong></td>
<td>525.0</td>
<td>17.0</td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td><strong>HASH-JOIN</strong></td>
<td>5056</td>
<td>178.5</td>
</tr>
<tr>
<td></td>
<td>4.1</td>
<td></td>
</tr>
</tbody>
</table>

Source: Prashanth Menon
REAL-WORLD IMPLEMENTATIONS

IBM System R
Oracle
Microsoft Hekaton
Cloudera Impala
MemSQL
VitesseDB
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
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.
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.
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 (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
```
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 (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.

\[
\text{SELECT } * \text{ FROM } A \text{ WHERE } A.id = 123
\]

\[
\text{SELECT } * \text{ FROM } A \text{ WHERE } A.id = ?
\]
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.

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

SELECT * FROM A
WHERE A.id = ?

SELECT * FROM A
WHERE A.id = 456
```
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
PARTING THOUGHTS

Query compilation makes a difference but is non-trivial to implement.
→ Speed-up always seems to be about 5-10x

The 2016 version of MemSQL is the best query compilation implementation out there. Hekaton is very good too.
The Art of Scan Sharing