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…
OBSERVATION

One way to achieve such a reduction in 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)  
);
QUERY PROCESSING

**Tuple-at-a-time**
→ Each operator calls `next` on their child to get the next tuple to process.

**Operator-at-a-time**
→ Each operator materializes their entire output for their parent operator.

**Vector-at-a-time**
→ Each operator calls `next` on their child to get the next chunk of data to process.

```sql
SELECT A.id, B.val
FROM A, B
WHERE A.id = B.id
AND B.val > 100
```
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

for t in A:
  emit(t)

for t in B:
  emit(t)

for t in C:
  emit(t)

for t in child.next():
  if evalPred(t):
    emit(t)

for t in child.next():
  if probe(t):
    emit(t1⨝t2)

for t1 in left.next():
  buildHashTable(t1)
  for t2 in right.next():
    if probe(t2):
      emit(t1⨝t2)

for t1 in left.next():
  buildAggregateTable(t1)
  for t in aggregateTable:
    emit(t)

for t1 in left.next():
  buildHashTable(t1)
  for t2 in right.next():
    if probe(t2):
      emit(t1⨝t2)
**PREDICATE INTERPRETATION**

**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)

```plaintext
TupleAttribute(B.val) = Parameter(0) + Constant(1)
```
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 (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)

**Tuple Attribute**:

- B.val

**Expressions**:

- `Parameter(0)`
- `Constant(1)`
- `1000 + Parameter(0)`
- `TupleAttribute(B.val) = 1000 + Parameter(0)`

**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)

**Predicate Interpretation**

```
TupleAttribute(B.val) = Parameter(0) + Constant(1)  
= 1000 + 999 + 1  
= 2000
```
**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**: (123, 1000)
- **Query Parameters**: (int:999)
- **Table Schema**:
  
  B → (int:id, int:val)

---

**Tuple Attribute**: B.val

- **Constant**: 1
- **Parameter**: \( \theta \)

\[
\text{true} = \text{TupleAttribute}(B.\text{val}) + \text{Constant}(1)
\]

\[
= 1000 + 1000 = 2000
\]

\[
= 999 + 1 = 1000
\]

\[
= \text{true}
\]
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

SQL Query

Parser

Binder

Optimizer

Compiler

System Catalog

Abstract Syntax Tree

Annotated AST

Physical Plan

Native Code

Cost Estimates
CODE GENERATION

Approach #1: Transpilation
→ Write code that converts a relational query plan into C/C++ 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 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
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.
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) + 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**

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 = ###
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.
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

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

Source: Konstantinos Krikellas
QUERY COMPILATION COST

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

Compilation Time (ms)

- **Q1**: Compile (-O0) - 121 ms, Compile (-O2) - 274 ms
- **Q2**: Compile (-O0) - 160 ms, Compile (-O2) - 403 ms
- **Q3**: Compile (-O0) - 213 ms, Compile (-O2) - 619 ms

Source: Konstantinos Krikellas
CMU 15-721 (Spring 2018)
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...
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

### Pipeline Boundaries

1. A.
2. B.
3. C.
4. A.id=C.a_id

\[ A \circledast B \circledast C \]
HYPER – J I T Q U E R Y C O M P I L A T I O N

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 △(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)
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>Query</th>
<th>HIQUE (ms)</th>
<th>HyPer (ms)</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
QUERY COMPILATION COST

LLVM's compilation time grows super-linear 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. Then execute that IR in an interpreter. Compile the query in the background. When the compiled query is ready, seamlessly replace the interpretive execution.
HYPER – ADAPTIVE EXECUTION

SQL Query

Optimizer (0.2 ms)

Query Plan

Code Generator (0.7 ms)

LLVM Passes (25 ms)

Byte Code Compiler (0.4 ms)

Unoptimized LLVM Compiler (6 ms)

Optimized LLVM Compiler (17 ms)

Byte Code

x86 Code

x86 Code

LLVM IR

LLVM IR

LLVM IR

LLVM IR
REAL-WORLD IMPLEMENTATIONS

IBM System R
Oracle
Microsoft Hekaton
Cloudera Impala
Actian Vector

MemSQL
VitesseDB
Apache Spark
Peloton
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.
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_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);
}
```
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 = ?
```
**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.

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

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

\[
\text{SELECT} \ast \text{ FROM A WHERE A.id = 456}
\]
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
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
APACHE SPARK

Introduced in the new Tungsten engine in 2015. The system converts a query's `WHERE` clause expression trees into ASTs. It then compiles these ASTs to generate JVM bytecode, which is then executed natively.
PELOTON

Full compilation of the entire query plan.

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

Use software pre-fetching to hide memory stalls.
PELOTON

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

Execution Time (ms)

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Interpreted</th>
<th>LLVM</th>
<th>LLVM + ROF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>901</td>
<td>892</td>
<td>383</td>
</tr>
<tr>
<td>Q3</td>
<td>1396</td>
<td>846</td>
<td>191</td>
</tr>
<tr>
<td>Q13</td>
<td>2641</td>
<td>1763</td>
<td>540</td>
</tr>
<tr>
<td>Q14</td>
<td>9960</td>
<td>21500</td>
<td>220</td>
</tr>
<tr>
<td>Q19</td>
<td>88147</td>
<td>87473</td>
<td></td>
</tr>
</tbody>
</table>

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. Hekaton is very good too.

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

Concurrency Control