Code Generation for Data Processing

Lecture 12: Query Compilation

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2022/23

Motivation: Fast Query Execution

- Databases are often used in latency-critical situations
  - Mostly transactional workload
- Databases are often used for analyzing large data sets
  - Mostly analytical workload; queries can be complex
  - Latency not that important, but through-put is
- Databases are also used for storing data streams
  - Streaming databases, e.g. monitoring sensors
  - Throughput is important; but queries often simple

Data Representation

- Relational algebra: set/bag of tuples
  - Tuple is sequence of data with different types
  - All tuples in one relation have same schema
  - Order does not matter
  - Duplicates might be possible (bags)
- Might have special values, e.g. `NULL`
- Values might be variably-sized, e.g. strings
- But: databases have *high* degree of freedom wrt. data representation

Query Plan

- Query often specified in “standardized format” (SQL)

- SQL is transformed into (logical) query plan
- Logical query plan is optimized
  - E.g., selection push down, transforming cross products to joins, join ordering
- Physical query plan
  - Selection of actual implementation for operators
  - Determine use index structures, access paths, etc.

Query Plan: Subscripts

- Query plan strongly depends on query
- Operators have query-dependent subscripts
  - E.g., selection/join predicate, aggregation function, attributes
  - Implementation of these also depends on schema
- Can include arbitrarily complex expressions
- Examples: $\bowtie^{HJ}_{s.matrnr=h.matrnr}$, $\sigma_{a.x<5\cdot(b.y-a.z)}$

Subscripts: Execution

- Option: keep as tree, interpret
  - + Simple, flexible
  - – Slow
- Option: compile to bytecode
  - + More efficient
  - – More effort to implement, some compile-time
- Option: compile to machine code
  - Code can be complex to accurately represent semantics
  - + Most efficient
  - – Most effort to implement, may need short compile-times

SQL Expressions

- Arithmetic expressions are fairly simple
  - Need to respect data type and check for errors (e.g., overflow)
  - Numbers in SQL are (fixed-point) decimals
- String operations can be more complex
  - `like` expressions
  - Regular expressions – strongly benefit from optimized execution
  - But: full-compilation may not be worth the effort
    often, calling runtime functions is beneficial
  - Support Unicode for increased complexity

Query Execution: Simplest Approach

$$\bowtie^{HJ}_{s.matrnr=h.matrnr}$$

studenten hoeren

- Execute operators individually
- Materialize all results after each operator
- “Full Materialization”

\+ Easy to implement
\+ Can dynamicnamically adjust plan
– Inefficient, intermediate results can be big

Iterator Model[51]

- Idea: stream tuples through operators
- Every operator implements set of functions:
  - `open()`: initialization, configure with child operators
  - `next()`: return next tuple (or indicate end of stream)
  - `close()`: free resources
- Current tuple can be pass as pointer or held in global data space
  - Possible: only single tuple is processed at a time

[51] G Graefe. "Volcano—an extensible and parallel query evaluation system". In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135.

Iterator Model: Example

```
struct TableScan : Iter {
  Table* table;
  Table::iterator it;
  void open() { it = table.begin(); }
  Tuple* next() {
    if (it != table.end())
      return *it++;
    return nullptr;
  } };
struct Select : Iter {
  Predicate p;
  Iter base;
  void open() { base.open(); }
  Tuple* next() {
    while (Tuple* t = base.next())
      if (p(t))
        return t;
    return nullptr;
  } };
```

```
struct Cross : Iter {
  Iter left, right;
  Tuple* curLeft = nullptr;
  void open() { left.open(); }
  Tuple* next() {
    while (true) {
      if (!curLeft) {
        if (!(curLeft = left.next()))
          return nullptr;
        right.open();
      }
      if (Tuple* tr = right.next())
        return concat(curLeft, tr);
      curLeft = nullptr;
    }
  }
};
```

- HashJoin builds hash table on first read; materialization might be useful

Iterator Model

- "Pull-based" approach
- Widely used (e.g., Postgres)
- Often have separate function for `first()` or rewind

+ Fairly straight-forward to implement
+ Avoids data copies, no dynamic compilation
– Only single tuple processed at a time, bad locality
– *Huge* amount virtual function calls

Push-based Model[52]

- Idea: operators push tuples through query plan bottom-up
- Every operator implements set of functions:
  - `open()`: initialization, store parents
  - `produce()`: produce items
    - Table scan calls `consume()` of parents
    - Others call `produce()` of their child
  - `consume()`: consume items from children, push them to parents
- Only one tuple processed at a time

[52] T Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *VLDB* 4.9 (2011), pp. 539–550.

Push-based Model: Example

```
struct TableScan {
  Table table;
  Consumer cons;
  void produce() {
    for (Tuple* t : table)
      cons.consume(t, this);
  }
};
struct Select {
  Predicate p;
  Producer prod;
  Consumer cons;
  void produce() { prod.produce(); }
  void consume(Tuple* t, Producer src) {
    if (p(t))
      cons.consume(t)
  }
};
```

```
struct Cross : Iter {
  Producer left, right;
  Consumer cons;
  Tuple* curLeft = nullptr;
  void produce() { left.produce(); }
  // Materializing one side might be better
  void consume(Tuple* t, Producer src) {
    if (src == left) {
      curLeft = t;
      right.produce();
    } else { // src == right
      cons.consume(concat(curLeft, t));
    }
  }
};
```

Push-based Model

- "Push-based" approach
- More recent approach

+ Fairly straight-forward, but less intuitive than iterator
+ Avoids data copies, no dynamic compilation
– Only single tuple processed at a time, bad locality
– *Huge* amount virtual function calls

Pull-based Model vs. Push-based Model[53]

- Two fundamentally different approaches
- Push-based approach can handle DAG plans better
  - Pull-model: needs explicit materialization or redundant iteration
  - Push-model: simply call multiple consumers
- Performance: nearly identical
  - Push-based model needs handling for limit operations
    otherwise table scan would not stop, even all tuples are dropped
- But: push-based code is nice after inlining

[53] A Shaikhha, M Dashti, and C Koch. "Push versus pull-based loop fusion in query engines". In: *Journal of Functional Programming* 28 (2018).

Pipelining

- Some operators need materialized data for their operation
  - Pipeline breaker: operator materializes input
  - Full pipeline breaker: operator materializes complete input before producing
- Other operators can be *pipelined* (i.e., no materialization)

- Aggregations
- Join needs one side materialized (pipeline breaker on one side)
- Sorting needs all data (full pipeline breaker)

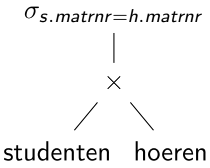
- System needs to take care of semantics, e.g. for memory management

Code Generation for Push-Based Model

- Inlining code in push-based model yields nice code

- No virtual function calls
- Producer iterates over materialized tuples and loads relevant data
  - Tight loop over base table – data locality
- Operators of parent operators are applied inside the loop
- Pipeline breaker materializes result (e.g., into hash table)

Code Generation: Example




```cpp
struct Query {
  Output out;
  Table tabLeft, tabRight;
  Tuple* curLeft = nullptr;
  void produce() {
    for (Tuple* tl : tabLeft) {
      curLeft = tl;
      for (Tuple* tr : tabRight) {
        Tuple* t = concat(curLeft, tr);
        if (t.s_matrnr == t.h_matrnr)
          out.write(t);
      }
    }
  }
};
```

How to Generate Code

- Code generator executes produce/consume methods
  - Method bodies don't do actual operations, but construct code
  - E.g., call `IRBuilder`
  - Call to helper functions for complex operations
    e.g. hash table insert/lookup, string operations, memory allocation, etc.

- Resulting code doesn't contain produce/consume methods
  only loops that iterate over data
  - No overhead of function calls

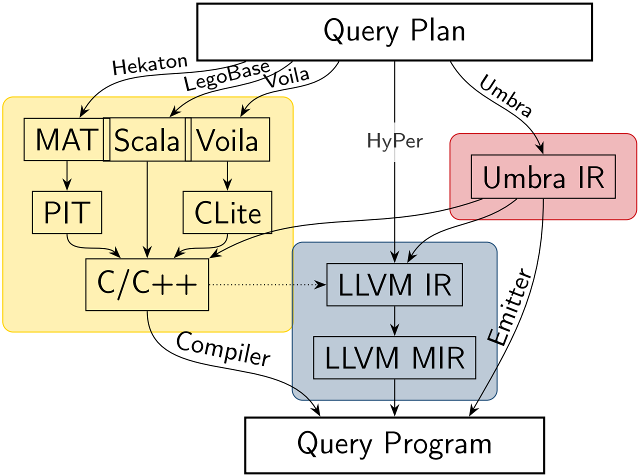
- Generate (at most) one function per pipeline
  - Allows for parallel execution of different pipelines

What to Generate

- Code generation allows for substantial performance increase
  - *Fairly* popular, even in commercial systems, despite engineering effort
  - Competence in compiler engineering is a problem, though

- Bytecode
  - Extremely popular: fairly simple, portable, and flexible
- Machine code through programming language (C, C++, Scala, ...)
  - Also popular: no compiler knowledge required, but compile-times are bad
- Machine code through compiler IR (mostly LLVM)
- Machine code through specialized IR (Umbra only)

What to Generate

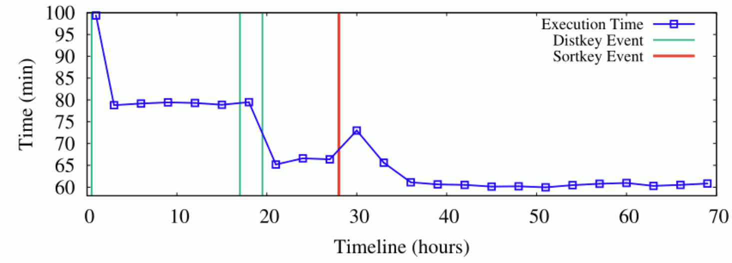


Case Study: Amazon Redshift[54]

“Redshift generates C++ code specific to the query plan and the schema being executed. The generated code is then compiled and the binary is shipped to the compute nodes for execution [12, 15, 17]. Each compiled file, called a segment, consists of a pipeline of operators, called steps. Each segment (and each step within it) is part of the physical query plan. Only the last step of a segment can break the pipeline.”

[54] N Armenatzoglou et al. “Amazon Redshift Re-invented”. In: *SIGMOD*. 2022.

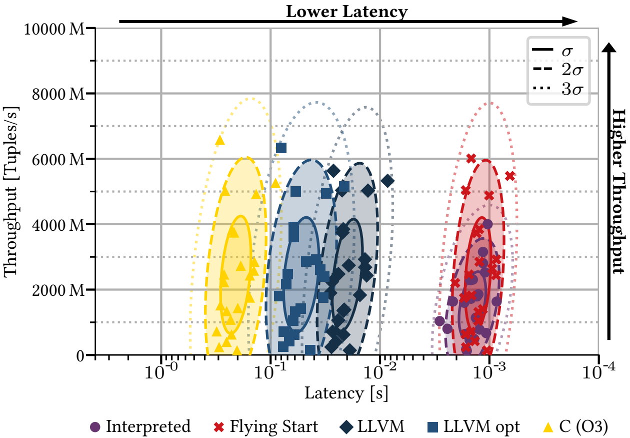
Case Study: Amazon Redshift[55]



"Figure 7(a) illustrates [...] from an out-of-box TPC-H 30TB dataset [...]. The TPC-H benchmark workload runs on this instance every 30 minutes and we measure the end-to-end runtime. Over time, more and more optimizations are automatically applied reducing the total work- load runtime. After all recommendations have been applied, the workload runtime is reduced by 23% (excluding the first execution that is higher due to compilation).

[55] N Armenatzoglou et al. "Amazon Redshift Re-invented". In: *SIGMOD*. 2022.

Compile Times: Umbra



TPC-H sf=30, AMD Epyc 7713 (64 Cores, 1TB RAM)

Vectorized Execution

- Problem: still only process single tuple at a time
- Doesn't utilize vector extensions of CPUs

- Idea: process multiple tuples at once
  - Also allows eliminating data-dependent branches, which not well-predictable
  - Esp. relevant when selectivity is between 10–90%

- Use of SIMD instructions requires column-wise store
  - Row-wise store would require gather operation for each load
  - Gather is very expensive

Vectorized Execution: SIMD Instructions

- Obvious candidate: initial selection over tables
  - Load vector of elements, use SIMD operations for comparison
  - Write back compressed result to temporary location for use in subsequent operations
  - Special compress instructions (AVX-512, SVE) highly beneficial
- Other operations much more difficult to vectorize
  - Initial hash table lookup requires gather; collisions difficult
  - When many elements are masked out, performance suffers

Vectorized Execution

- Bytecode interpretation substantially benefits from vectorized execution
- Key benefit: less dispatch overhead
- Typically much larger “vectors” (>1000)

- Comparison with non-vectorized machine code generation:
  - Vectorization often beneficial for initial scan
  - Code generation is faster than bytecode-interpred vec. execution
  - But: a good vectorized engine is not necessarily *slow*
- Vectorized execution probably more popular than code generation

Query Compilation – Summary

- Databases have trade-off between low latency and high throughput
- Evaluation needed for operators and subscripts
- Subscripts easy to compile
- Operator execution: full materialization vs. pipelined execution
- Pull-based vs. push-based execution
- Push-based allows for good code generation
- Bytecode and programming languages are widely used in practice
- Vectorized execution improves performance without native code gen.

Query Compilation – Questions

- Why are low compile times important for databases?
- What is the difference between push-based and pull-based execution?
- Why does push-based execution allow for higher performance?
- How to generate code for a query?
- How does vectorized execution improve performance?
- Why do many database engines not use machine code generation?