Towards Just-in-time Compilation of SQL Queries with OMR JitBuilder

Debajyoti Datta
University of New Brunswick
Fredericton, Canada
debajyoti.datta@unb.ca

Mark Stoodley
IBM Canada
Markham, Canada
mstoodle@ca.ibm.com

Suprio Ray
University of New Brunswick
Fredericton, Canada
sray@unb.ca

ABSTRACT

The evaluation of SQL expressions and tuple materialization can consume a significant portion of the overall execution time of a query. The goal of our work is to generate efficient machine code for scan, filter, join and group-by operations for a given SQL expression by Just-in-time (JIT) compilation using the OMR JitBuilder compiler framework. Our approach creates a blend of specialized code consisting of compile-time constants and JIT computation for parts of the same SQL expression. The implementation is based on a light-weight integration of JitBuilder into PostgreSQL 12.5, where both the JIT compiled code and interpreted evaluation co-exist for different opcodes in the same bytecode interpreter. We demonstrate with our enhanced PostgreSQL 12.5 that our approach offers improved query performance over purely interpreted execution.

1 INTRODUCTION

In recent years, the topic of query compilation has become prominent, particularly for analytical workloads involving long running queries. Query compilation can be more efficient than the tuple-at-a-time processing model of Volcano-style [6] query execution. This involves translating an SQL query into machine code using a compiler framework, such as LLVM [10]. Recent research on query compilation focused on aspects, such as compilation of execution plans [11], and ameliorate the overhead of compilation time [8]. However, the integration of these techniques within a database system requires significant effort to re-architect the query engine. As a result, well-established database systems, such as PostgreSQL, still do not support full-fledged compiled plan generation and execution. Hence, it may be easier to focus on incremental adoption of query compilation techniques. We argue that SQL expression and tuple materialization are "low-hanging fruits" as targets for Just-in-time (JIT) compilation, as this does not require overhauling the query engine completely, as demonstrated by Butterstein and Grust [3]. However, this has not yet received much attention from the research community.

Expressions are prevalent in SQL queries, which may appear in different parts of a query, including, `select` clause, `where` clause, invocation of aggregates, and grouping for group-by. We profiled the execution of a TPC-H benchmark [1] query Q6, which is shown in Figure 1. Our profiling provides ample evidence that a significant percentage of PostgreSQL query execution time is spent in the function `ExecInterpExpr`. A detailed discussion regarding PostgreSQL expression evaluation and our findings are presented in Section 2.1. The `ExecInterpExpr` function is responsible for SQL expression evaluation and tuple materialization among other features. Therefore, the focus of this work is to try to speed up SQL expression evaluation and tuple materialization to gain runtime improvement in query execution by performing JIT compilation of parts of the query where most of the execution time is spent. This compilation process is relatively fast. JIT compilation converts interpreted code into native code and removes the overhead of interpretation, specializing code for constant arguments, and reducing the number of branches and indirect jumps/calls.

In this paper we show the benefits of our approach with our enhanced PostgreSQL. We adopt a non-invasive approach similar to that of Butterstein and Grust [3], which does not require any change to PostgreSQL's Volcano-style 'interpreted' processing model of query execution. However, unlike their approach, we use a novel compiler framework, OMR JitBuilder [5] to compile SQL expressions and tuple materialization (deformation). Our lightweight JIT compilation approach takes advantage of JitBuilder's support for simple and flexible APIs for runtime code generation. We integrate our JIT compilation approach within PostgreSQL version 12.5. Our experimental evaluation involving TPC-H queries with three scale factors show significant performance improvement over the original (pure interpretation based) PostgreSQL execution.

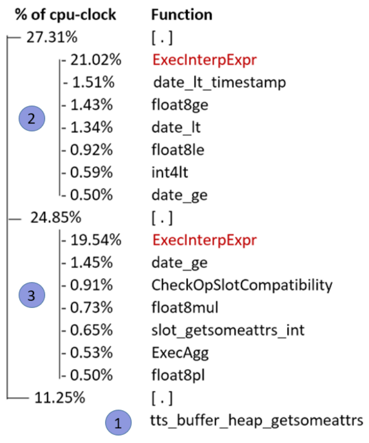
2 QUERY PROCESSING IN POSTGRESQL

In this section, we describe how PostgreSQL processes an SQL query and how expressions are evaluated.

2.1 PostgreSQL backend process

When PostgreSQL is launched, a background worker process is created, which basically handles all queries issued by the connected clients. This backend processing consists of five phases, as shown in Figure 2. In the first phase, the parser generates a parse tree from an SQL statement in plain text, which is then fed to the analyzer to perform a semantic analysis of a parse tree and generate a query tree. If the semantics of the incoming query is correct, the rewriter checks for any SQL view statement and the specification of a view. For this purpose the rewriter implements a rule system. In the third phase, the planner generates a plan tree or query execution plan from the query tree. In the fourth phase, the executor uses the plan tree to process the query. In PostgreSQL the plan tree generated by the planner is a collection of nodes, where each node consists of a series of steps required to evaluate its implementation. Finally, the

CASCON 2021, November 22 - 26, 2021, Online/Virtual



```
% of cpu-clock    Function
— 27.31%          [ . ]
   - 21.02%       ExecInterpExpr
   - 1.51%        date_lt_timestamp
   - 1.43%        float8ge
 ② - 1.34%        date_lt
   - 0.92%        float8le
   - 0.59%        int4lt
   - 0.50%        date_ge
— 24.85%          [ . ]
   - 19.54%       ExecInterpExpr
   - 1.45%        date_ge
   - 0.91%        CheckOpSlotCompatibility
 ③ - 0.73%        float8mul
   - 0.65%        slot_getsomeattrs_int
   - 0.53%        ExecAgg
   - 0.50%        float8pl
— 11.25%          [ . ]
              ①   tts_buffer_heap_getsomeattrs
```

Figure 1: TPC-H Q6: profiled with perf (scale factor 4)

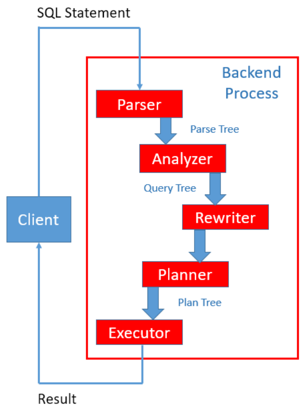
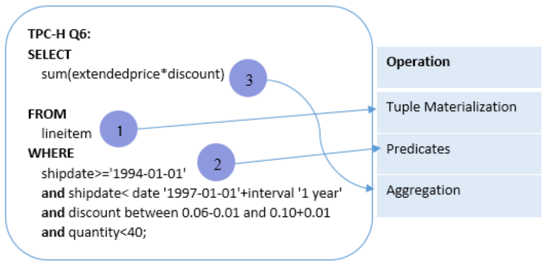


Figure 2: Query processing in PostgreSQL

executor executes the query as determined by the plan tree and the result is returned to the client.

2.2 Expression evaluation in PostgreSQL

Figure 3 shows the TPC-H benchmark query Q6 and a breakdown of the different operations that are part of the query plan. The query consists of a conjunction of Boolean filters in the *where* clause marked as ②. Indicator ① shows tuple materialization, whose purpose is to extract the underlying tuples of the table in the in-memory buffer pool after the corresponding pages have been retrieved from disk. The tuples are stored in a representation, which is relatively compact and the goal of tuple materialization is to convert this into a form that can be accessed more efficiently and conveniently. This enables the tuple attributes to be ready for operations such as expression evaluation, aggregation or join. The ② + ① form the sequential scan node consist of a PostgreSQL `qual` (qualification conditions in a predicate) list and is initialized with a set of Boolean and arithmetic expressions, which are evaluated using back-end functions such as `date_ge`, `float8le`, `int4lt`, etc. During evaluation, if a tuple qualifies the `qual` list, it can proceed to the aggregation, which is the sum of multiplication of the attributes `extendedprice` and `discount`.



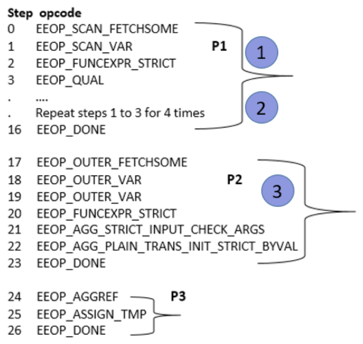
```
TPC-H Q6:
SELECT
  sum(extendedprice*discount)  ③
FROM
  lineitem  ①
WHERE  ②
  shipdate>='1994-01-01'
  and shipdate< date '1997-01-01'+interval '1 year'
  and discount between 0.06-0.01 and 0.10+0.01
  and quantity<40;
```

Figure 3: TPC-H Q6

To determine the impact of SQL expression evaluation on the overall query performance in PostgreSQL, we used the Linux tool perf [2] to instrument and profile query Q6. Figure 1 shows a breakdown of the execution time for Q6. It is evident that (27.31 + 24.85)% = 52.16% of the overall execution time is spent in the function `ExecInterpExpr` and a further 11.25% of the time is spent in `heap_getsomeattrs`. The function `ExecInterpExpr` lies at the core of expression evaluation, whereas `heap_getsomeattrs` performs tuple materialization. Hence, these two features together consume a substantial (52.16 + 11.25)% = **63.41%** of the query execution time. Note the two variants of time consumption in the Figure 1 for the same function `ExecInterpExpr` correspond to different control flow paths within this function.

The characteristics of this distribution of execution time encourages us to take a deep dive into `ExecInterpExpr` revealing PostgreSQL internals. A plan node in PostgreSQL may contain an expression tree representing a target list, qualification conditions and others. Each expression tree is represented by `ExprState` nodes. To prepare to execute each expression tree, the tree is converted into a linear sequence of opcodes, where each opcode represents a unit of operation to be performed. These opcodes include fetching an attribute, calling the backend implementation of operators and functions, evaluating the qualification of a tuple, etc. There are a total of 90 opcodes in and for every `ExprState` node, the corresponding opcode sequence is maintained in the `ExprEvalStep` array. Finally, the `ExprState` is prepared for interpreted execution where: a) the interpreter is setup b) the function to be called for evaluating the current `ExprState` node `(ExprState->evalfunc)` is set to `ExecInterpExpr()`, which is a computed goto based interpreter. During interpreted execution, `ExecInterpExpr()` iterates over the opcode sequence attached to an `ExprState` node and executes the corresponding implementation for each opcode.

Figure 4 shows the different sequences of opcodes generated for Q6, where the first two sets involve the most recurring sequences and the last set is evaluated only once for Q6. It can also be seen that there are 17 opcodes in a sequence to evaluate the predicate, which is referred as phase P1. Step 0: `EEOP_SCAN_FETCHSOME` is used to scan `lineitem` table to get the desired attributes, which is then loaded to a transient variable in step 1 by implementing the opcode `EEOP_SCAN VAR`. Next, in step 2: `EEOP_FUNCEXPR_STRICT` is used to call the back-end function of the corresponding boolean operator in the filter condition. For example, the back-end function `int4lt` is used to evaluate the operator < in the filter condition `quantity < 40`. Finally, in step 3 the opcode `EEOP_QUAL` checks if the current filter condition was satisfied. Steps 1, 2, and

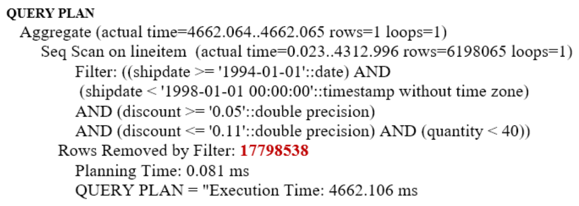


```
Step  opcode
0     EEOP_SCAN_FETCHSOME
1     EEOP_SCAN_VAR                          P1   1
2     EEOP_FUNCEXPR_STRICT
3     EEOP_QUAL
.     ....                                        2
.     Repeat steps 1 to 3 for 4 times
16    EEOP_DONE

17    EEOP_OUTER_FETCHSOME
18    EEOP_OUTER_VAR                         P2   3
19    EEOP_OUTER_VAR
20    EEOP_FUNCEXPR_STRICT
21    EEOP_AGG_STRICT_INPUT_CHECK_ARGS
22    EEOP_AGG_PLAIN_TRANS_INIT_STRICT_BYVAL
23    EEOP_DONE

24    EEOP_AGGREF                            P3
25    EEOP_ASSIGN_TMP
26    EEOP_DONE
```

Figure 4: TPC-H Q6: opcode sequence in 3 phases



```
QUERY PLAN
  Aggregate (actual time=4662.064..4662.065 rows=1 loops=1)
     Seq Scan on lineitem  (actual time=0.023..4312.996 rows=6198065 loops=1)
          Filter: ((shipdate >= '1994-01-01'::date) AND
           (shipdate < '1998-01-01 00:00:00'::timestamp without time zone)
          AND (discount >= '0.05'::double precision)
          AND (discount <= '0.11'::double precision) AND (quantity < 40))
        Rows Removed by Filter: 17798538
          Planning Time: 0.081 ms
          QUERY PLAN = "Execution Time: 4662.106 ms
```

Figure 5: TPC-H Q6: explain analyze output

3 are repeated until all the `qual` list have been traversed and if the conjunction of the filters is true, the rows are qualified to be aggregated in phase P1. If a row satisfies the `qual` list, it is projected to the next phase of `ExprState`, where the operation `extendedprice * discount` is calculated and the result is stored to the aggregation state represented by the data structure `AggStatePerGroupData`. Finally, when all the tuples are evaluated, the third opcode sequence, phase P3, is used to store the result into the `resultslot` field of `ExprState`.

Figure 5 shows the output of `EXPLAIN ANALYZE` for Q6. With a scale factor 4, the `lineitem` table consists of 24 million rows. Overall, expression evaluation and tuple materialization together account for (27.31 + 24.85 + 11.25)% = 63.41% of the query execution time. This is a significant portion of the overall execution time, and is primarily due to repeated interpretation of the same code.

3 JITBUILDER COMPILER FRAMEWORK

JitBuilder is a library developed in the Eclipse OMR [4] project. It was developed to simplify the tasks for a runtime system to incorporate a JIT compiler. JitBuilder provides a simple lifecycle API `initializeJit()` and `shutdownJit()`, as well as a simple descriptive API for the runtime system to describe what code needs to be generated at runtime. With a few hundred to a few thousand lines of code, a JIT compiler can be implemented that automatically targets multiple platforms (including X86, POWER, IBM Z, AArch64, and RISC-V).

We illustrate with an example to show how JitBuilder can be used to dynamically generate a function that can multiply the corresponding elements of two vectors together into a result vector. First, we subclass JitBuilder's `MethodBuilder` class and describe the function parameters and return value type in the constructor as shown in Figure 6. This function is given a name "multiply" and

```
using namespace OMR::JitBuilder;
class Multiply : public MethodBuilder {
   Multiply::Multiply(TypeDictionary *types,
                      IlType *elementType)
   : MethodBuilder(types, elementType )
   , _T(types->PointerTo(elemType))
   {
      DefineName("multiply");
      DefineParameter("rslt", _T);
      DefineParameter("vec1", _T);
      DefineParameter("vec2", _T);
      DefineParameter("len", Int32);
      DefineReturnType(type);
   }
   virtual bool buildIL();
   protected:
   IlType *_T;
};
```

Figure 6: `Multiply` example parameters and return type

```
bool Multiply::buildIL() {
   IlValue *rslt = Load("rslt");
   IlValue *vec1 = Load("vec1");
   IlValue *vec2 = Load("vec2");
   IlValue *len =  Load("len");
   IlValue *zero = ConstInt32(0);
   IlValue *one =  ConstInt32(1);

   IlBuilder *lp = OrphanBuilder();
   ForLoopUp("i", &lp, zero, len, one); {
      IlValue *i, *v1, *v2, *prod;
      i = lp->Load("i");
      v1 = lp->ArrayLoadAt(_T, vec1, i);
      v2 = lp->ArrayLoadAt(_T, vec2, i);
      prod = lp->Mul(v1, v2);
      lp->ArrayStoreAt(T, rslt, i, prod);
   }
   Return();
   return true;
  }
```

Figure 7: Slightly simplified `Multiply::buildIL()`

it takes four parameters: three pointers (one for the result vector and two for the input vectors) to the vector element type (which can be chosen dynamically as a parameter to the constructor) and the last parameter specifies the length of each of the vectors. To describe what the generated function should do when called, one overrides the virtual `MethodBuilder` function `buildIL`, as shown in Figure 7 in a slightly simplified form (`ArrayLoadAt` and `ArrayStoreAt` each represent a sequence of two operations to index into then to load or store an array element in JitBuilder).

The code in Figure 7 demonstrates a few key features of the JitBuilder API. First, JitBuilder uses *`IlBuilder`* objects to represent the code paths needed in the generated code (a `MethodBuilder` object itself corresponds to the compiled function's entry point). Another code path, representing the body of the loop that iterates through the input vectors, is created via `OrphanBuilder()` and used later as `lp`. Second, operations are described in order as they should be executed in a code path. When the generated code is called, for example, the `rslt` parameter will be loaded first, then the `vec1` parameter, then `vec2`, and so on. More complex operations can also be added to the sequence, like `IfThenElse` to introduce conditional code paths or `ForLoopUp` to iterate a number of times over a set of operations (specified by a code path). These more complex operations simplify the job of generating code dynamically by taking care of the details required to implement a for loop or to lay out the code for an if-then-else diamond. In this example, the body of the for loop will load an element indexed by `i` from `vec1`, then another element, also indexed by `i` from `vec2`. It then multiplies them together using `Mul`, and finally stores the product into the `rslt` vector, indexing by `i`. In this case, no more

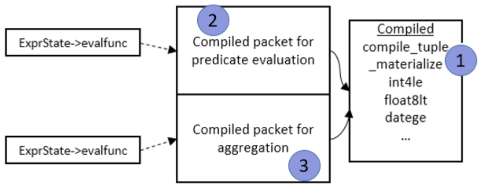

ExprState->evalfunc
ExprState->evalfunc
2
Compiled packet for predicate evaluation
Compiled packet for aggregation
3
Compiled
compile_tuple_materialize
int4le
float8lt
datege
...
1

Figure 8: Compiling the 'hot' phases

complex operations are added to the loop body, but arbitrary complexity can be achieved by creating more `IlBuilder` objects and using additional operations like `IfThenElse` or `ForLoopUp` inside the loop. The final `Return` call will execute after the entire `ForLoopUp` operation completes, causing the generated function to return to its caller.

To compile the code described by the `Multiply` class, one creates a `Multiply` object and calls `compileMethodBuilder` on it to dynamically compile its code and provide an entry point to that code, typically within a few milliseconds. That entry point can then be called like a C function. Note that the base type of the vector elements never appears directly in the `buildIL()` code, so this one `Multiply` class can be used, analogous to how C++ templates work in source code, to generate code at runtime to handle whichever primitive vector type is needed simply by creating different `Multiply` objects and passing a specific element type (like `Double` or `Int32`) to the constructor. One can also write arbitrary conditional code directly into `buildIL()` so that different code can be generated at will. For example, the `buildIL` function can switch over the different byte code types of a language runtime or runtime system so that it can generate code for a specific sequence of those byte codes. JitBuilder also includes facilities, like `BytecodeBuilders` and `VirtualMachineOperandStack` specifically designed to help generate efficient compiled methods for bytecode based runtimes. We use some of these facilities in our framework to compile opcode sequences.

4 OUR APPROACH

Section 2 shows how different parts of a query in PostgreSQL are bottlenecked due to CPU intensive code of comparatively smaller size, which burns a large number of CPU cycles. As the code has to be very generic to evaluate random SQL expressions on random tables, it often leads to large number of indirect jumps, unpredictable branches and an oversized instruction set for a particular operation. Compiling this bottleneck prone code at run-time would generate native code by which a substantial amount of indirect jumps can be removed. This can be achieved by identifying the compile-time constants such as table schema information and directly evaluating the branch at compile-time, which entirely removes the branch. Another way is to convert the indirect branches into direct branches. For example, replacing the call to the SQL operator's backend implementation with a direct call to the compiled function.

We propose an iterative approach to adopt compilation-based evaluation of SQL queries. Our implementation is non-invasive, as it does not alter the computed `goto`-based expression evaluation of PostgreSQL and in this way we formulate a hybridized model consisting of both compilation and interpretation techniques by utilizing run-time code specialization opportunities. Each of the following sub-sections expands on our approach.

4.1 Type Synchronization

In order to execute the operations in an SQL query, it becomes inevitable to translate the PostgreSQL specific data types to our compiler framework. Our JitBuilder framework can only handle primitive data types. For the data types that are not defined in the JitBuilder framework, but exist in PostgreSQL and are also needed for some operations, Jitbuilder provides an elegant way to create new data types that will be automatically translated by the compiler. To synchronize these data types we create a `HandleNewDataType` class, which inherits the `TypeDictionary` class of the OMR compiler framework. In this class we define PostgreSQL specific data types such as `Datum`, struct types such as `FormData_pg _attribute` to hold the schema information, etc. In order to intimate the generated code about the new data types, we simply create an object of the `HandleNewDataType` and pass it to the `MethodBuilder` responsible for generating code.

4.2 Compiling the hot phases

In our approach, we compile the opcode sequence for each `Expr-State` node and replace the pointer to the PostgreSQL interpreter function `ExecInterpExpr` with an entry to the compiled opcode sequence. To achieve it, a new function `EntryToCompiledExpr` is created, which when receiving a node compiles it if this was the first call to the given `ExprState` node. Otherwise, the opcode sequence must have been compiled and stored into a handle to the compiled code as shown in Figure 8. The proposed compiled evaluation utilizes JIT compilation-based on JitBuilder.

We take advantage of the code specialization afforded by JitBuilder. The development of our proposed system is dependent on JitBuilder's `BytecodeBuilder` objects and related C++ client APIs to implement a `while+switch` based implementation to evaluate the opcode sequences. We create one `BytecodeBuilder` object for each `ExprEvalOp` step, with that object responsible for generating the code specific to that opcode. After the bytecode objects are created, the process of walking through the bytecodes in a proper order is initiated to inject the operations for each bytecode handler into the `MethodBuilder` object using JitBuilder's handy `BytecodeBuilder` worklist.

JitBuilder is equipped with a variety of APIs that allow us to perform `Load() and Store()` on primitives, variables, structs, unions, etc. Building a JIT compiler for expression compilation using JitBuilder involves the following steps: **1)** *define the compiled method:* by inheriting the `MethodBuilder` object of JitBuilder we **a)** declare the name of the compiled method **b)** define each native datatype of PostgreSQL to `Iltype` so that compiled code knows the corresponding JitBuilder types and **c)** define the return type of the compiled code, and finally **2)** *define logic that the compiled code should follow* by implementing the virtual function of the `MethodBuilder`.

4.3 Specializing attribute access

Accessing the attributes is one of the most important parts of evaluating a query, which enables the tuple attributes to be ready for

```
for (attnum = 0; attnum < natts; attnum++)
{
        Form_pg_attribute thisatt = att[attnum];
        if (att_isnull(attnum, bp))
        {
                values[attnum] = (Datum) 0;
                isnull[attnum] = true;
                continue;
        }
        isnull[attnum] = false;
        off = att_align_nominal(off, thisatt->attalign);
        values[attnum] = fetchatt(thisatt, tp + off);
        off = att_addlength_pointer(off, thisatt->attlen, Tp + off);
}
```

Figure 9: Specializing the attribute access using loop-unroll

operations such as expression evaluation, aggregation or join. The tuple attributes are stored in `TupleDescData` struct in the shared buffers of the PostgreSQL application. This struct also contains the schema information for a particular attribute such as the data type, the size and offset. Every tuple has an array of attributes associated with it, where each attribute is of type `Form_pg_attribute` struct. In order to access the attributes, we need to traverse the array using a proper offset value at any attribute. In the interpreted evaluation the schema information is evaluated every time an attribute is fetched. Since, schema information will be same for a given table throughout the lifetime of the query execution, we can evaluate the same at the compile-time only once thus avoiding a large number of unnecessary branching, indirect calls and also reducing the amount of information to be evaluated multiple times. This approach lets us generate highly efficient code at run-time. Because evaluating code at compile-time reduces the amount of code to be generated at run-time, which in turn helps reduce compilation time. In Figure 9 the code marked in red can be easily computed at compile-time, which is achieved by leveraging loop-unrolling to evaluate the loop at compile-time as the number of attributes represented by `natts` is constant to this code. Similarly, the offset is also computed at compile-time. In this way our approach naturally extends to specialize the code by determining the type and alignment of each attribute of a row from the schema during compile-time. Based on this approach we can speed up the process of calculating the offset needed to move to the next attribute in a row which results in fast access to attributes as compared to `slot_getsomeattrs_int()`.

4.4 Compiling the backend functions

Since PostgreSQL is an object relational model-based system, it defines an independent implementation for the SQL operators and at the same time gives us the flexibility to add our own implementation for operators. We take this opportunity to compile the back-end functions such as `int4lt`, `date_gt`, `int4eq`, etc., which are used to evaluate operations: <, >, =, *, etc. In this way we are able to replace a large number of indirect calls with a direct call to the compiled function for the corresponding operator. We leverage the concepts of Constant Propagation to specialize the code for these functions. For example, in Figure 10 one of the arguments is constant input to `int4eq` function. This argument can be treated as a compile-time constant throughout the lifetime of the query and can be directly evaluated during compile-time.

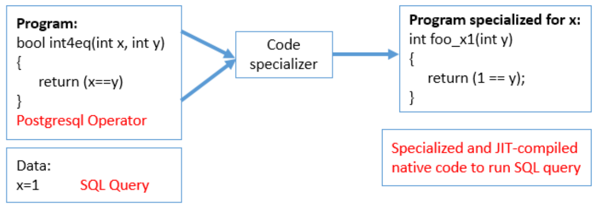


Figure 10: Specializing SQL operator implementation

```
Lpredicate:
# Compare shipdate to -2191 (encoded 1994-01-01 value)
                cmp qword ptr [r10+0x50], 0xfffffffffffff771
                jge Label Lpass

# gets here if predicate fails
# set predicate result (eax) to 0
                xor eax, eax
# Store 0 to ExprState.resvalue
                mov qword ptr [r8+0x8], 0x00000000
Leval_result:
# test if result is zero or should continue
                test        eax, eax
                jne Label Lcontinue

# this predicate failed, update output, return
        ...
        ret
Lpass:
# gets here if row passes predicate
# Store 1 to ExprState.resvalue
                mov qword ptr [r8+0x8], 0x00000001
# Store 0 to ExprState.resnull
                mov byte ptr [r8+0x5], 0x00
# set predicate result (eax) to 1
                mov eax, 0x00000001
                jmp Label Leval_result

Lcontinue: # evaluate more predicates...
```

Figure 11: Instructions for `shipdate >= '1994-01-01'`

Our code specializations and refinements target the hot execution zones, such as `ExecInterpExpr`, which consumes more than 60% of the time for e.g. Q6. To avoid any performance loss due to the necessary compilation time, substantial effort is applied to trace the frequently occurring opcode sequences and also sequences that are less likely to occur. For instance, in Q6 the last sequence occurs only once per query (see Figure 4), and hence this is left to be evaluated by the interpreter. Thus, we formulate a hybrid model consisting of both compilation and interpretation techniques.

4.5 Code generation illustration

Our implementation is based on an extended version of PostgreSQL 12.5. This is enhanced with JIT compilation of SQL expression and tuple materialization using OMR JitBuilder. The JIT compilation process generates specialized instruction sequences like that for an example filter predicate `shipdate >= '1994-01-01'` shown in Figure 11. There are several improvements we can make to this sequence to further reduce path length, but it already represents a significantly shorter instruction sequence than that the interpreter executes to apply the same filter.

5 EVALUATION

The evaluation experiments are conducted on a machine with Intel Xeon Gold 5120 processors, with a total of 56 physical cores, having an aggregate memory of 128 GB RAM. The PostgreSQL version is 12.5 and the source code is compiled using g++ (version 7.5.0) with

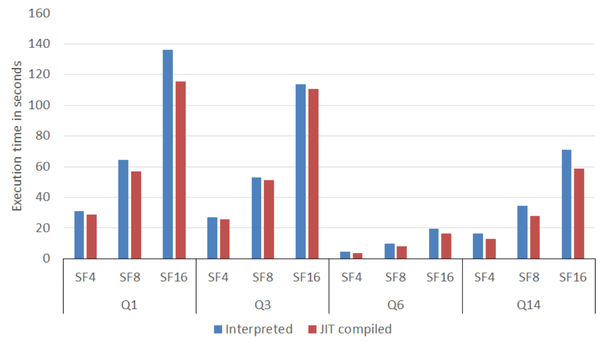


Figure 12: Execution time for interpreted vs. JIT compiled (TPC-H Q1, Q3, Q6, Q14 scale factors 4, 8 and 16)

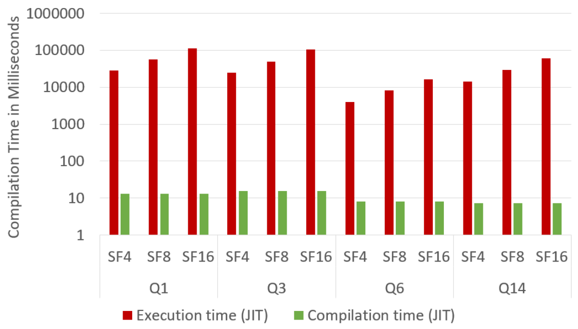
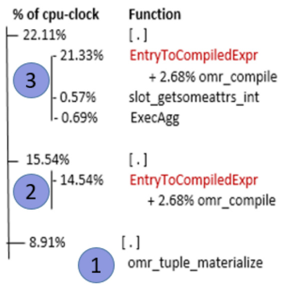


Figure 13: JIT compiled: execution time vs. compilation time comparison (TPC-H Q1, Q3, Q6, Q14 scale factors 4, 8 and 16)


```
% of cpu-clock    Function
— 22.11%          [.]
    21.33%        EntryToCompiledExpr
                     + 2.68% omr_compile
  3 0.57%         slot_getsomeattrs_int
    0.69%         ExecAgg

— 15.54%          [.]
  2 14.54%        EntryToCompiledExpr
                     + 2.68% omr_compile

— 8.91%           [.]
                1 omr_tuple_materialize
```

Figure 14: TPC-H Q6: profiled with perf after compilation (scale factor 4)

-O3 flag. We use 4 queries from the TPC-H benchmark: Q1, Q3, Q6, and Q14. These queries are evaluated for 3 dataset sizes (TPC-H scale factors): 4, 8 and 16. Each query is run 12 times and we report the average execution time of the last 10 query runs.

As shown in Figure 12, our approach performs better than the interpreted execution in all cases. In Figure 13, we compare the execution time with compilation time of our approach (JIT compiled). In each case the compilation time is significantly small compared to the execution time. For instance, the compilation time constitutes 0.01% of the query latency (compilation + execution time) for Q14 at scale factor 16. Hence, the compilation time overhead is not substantial. Our profiling of Q6 (Figure 14) suggests that compiling SQL expression and tuple materialization can speed up query execution. In Q6 the amount of time spent in predicate evaluation alone gained 13% speedup over interpreted execution, whereas a descent speedup was also recorded for ① and ③ with 2.3% and 1.5% respectively.

6 RELATED WORK

Advances in hardware technology, leading to growing memory capacity, have enabled main memory databases to be commercially viable. As a result, in recent years query compilation has become an active area of research. The primary focus of some of this research has been the compilation of query execution plans. Query compilation can address some of the limitations of classical iterator style processing techniques [6] that can lead to poor performance, such as the lack of data locality, frequent instruction miss-prediction, and high branching. Krikellas et al. [9] proposed a template based approach for code generation from SQL query plans. Neuman et al. [11] proposed an approach to compile query plans into more efficient machine code using LLVM compiler framework. A recent work by Kersten et al. [7] focuses on minimizing compilation time.

Due to challenges associated with re-architecting query engines, most of the commercial databases have yet to adopt full-fledged compilation-based query execution. Consequently, an incremental approach is preferable, in which a part of the query plan is adaptively compiled into machine code. To that end, SQL expression and tuple materialization are ideal targets for compilation. The work that is the most similar to ours is that of Butterstein et al. [3], which uses LLVM framework. Unlike their work, we utilize a novel compiler framework, OMR JitBuilder [5], which is a light-weight JIT framework and supports flexible APIs for runtime code generation.

7 CONCLUSION

We proposed an iterative approach to JIT compile SQL expression and tuple materialization, which constitue a substantial part of the overall query execution time. Our approach is a hybrid execution model consisting of both compilation and interpretation. Our JIT compilation approach is based on OMR JitBuilder, and is integrated within PostgreSQL 12.5. Experimenal evaluation with TPC-H benchmarks demonstrate that our approach can yield significant performance improvement over pure interpretation based SQL query execution.

REFERENCES

[1] 2019. TPC-H benchmark specification 2.18.0_rc2.
[2] 2021. perf: Linux profiling with performance counters.
[3] Dennis Butterstein and Torsten Grust. 2016. Precision Performance Surgery for PostgreSQL: LLVM—Based Expression Compilation, Just in Time. *Proc. VLDB Endow.* 9, 13, 1517–1520.
[4] Eclipse OMR 2021. Eclipse OMR. https://github.com/eclipse/omr.
[5] Eclipse OMR JitBuilder 2021. JitBuilder release notes and code samples. https://github.com/eclipse/omr/tree/master/jitbuilder/release.
[6] G. Graefe and W. J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *ICDE*. 209–218.
[7] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* (jun 2021).
[8] A. Kohn, V. Leis, and T. Neumann. 2019. Making Compiling Query Engines Practical. *TKDE* 33, 2 (2019), 597–612.
[9] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. 613–624.
[10] LLVM [n.d.]. The LLVM Compiler Infrastructure. https://llvm.org/.
[11] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550.