VLDB Journal manuscript No.
(will be inserted by the editor)

Low-Latency Query Compilation

Henning Funke · Jan Mühlig · Jens Teubner

the date of receipt and acceptance should be inserted later

Abstract Query compilation is a processing technique that achieves very high processing speeds but has the disadvantage of introducing additional compilation latencies. These latencies cause an overhead that is relatively high for short-running and high-complexity queries. In this work, we present *Flounder IR* and *ReSQL*, our new approach to query compilation. Instead of using a general purpose intermediate representation (e.g. LLVM IR) during compilation, ReSQL uses Flounder IR, which is specifically designed for database processing. Flounder IR is lightweight and close to machine assembly. This simplifies the translation from IR to machine code, which otherwise is a costly translation step. Despite simple translation, compiled queries still benefit from the high processing speeds of the query compilation technique. We analyze the performance of our approach with micro-benchmarks and with ReSQL, which employs a full translation stack from SQL to machine code. We show reductions in compilation times up to two orders of magnitude over LLVM and show improvements in overall execution time for TPC-H queries up to 5.5× over state-of-the-art systems.

Henning Funke
TU Dortmund University
E-mail: henning.funke@cs.tu-dortmund.de

Jan Mühlig
TU Dortmund University
E-mail: jan.muehlig@cs.tu-dortmund.de

Jens Teubner
TU Dortmund University
E-mail: jens.teubner@cs.tu-dortmund.de

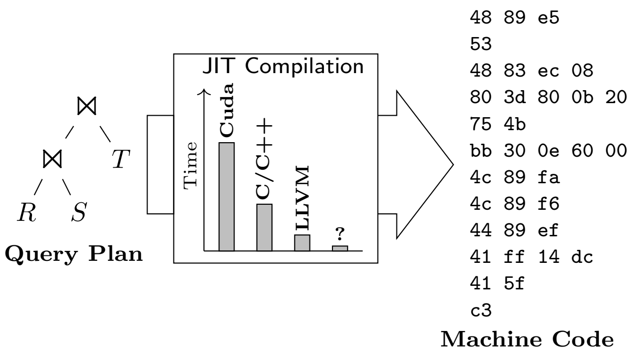


Fig. 1 Effect of different intermediate representation levels on JIT query processing performance.

1 Introduction

Query compilation is a technique for query execution with extremely high efficiency. It uses *just-in-time* (JIT) compilation to generate custom machine code for the execution of every query. The approach leverages a compiler stack that first translates the query from a relational query plan to an *intermediate representation* (IR), and then from the IR to *native machine code* for the target machine. The execution-efficiency of the compiled code is very high compared to standard interpretation-based backends. However, by using compilation the technique adds a step to query execution, which introduces translation cost. Especially short-running queries and queries with high complexity experience a relatively high translation cost, which ultimately extends query response times.

When using query compilation for queries on smaller datasets, the relative cost of compilation increases. The query engine spends most of its time on compilation before entering execution only for a very short time. Further, complex queries can have particularly long compilation times due to complexity of algorithms used in JIT machine code translation [30]. Approaches to mitigate the impact of compilation time on response time have been proposed previously [19]. However, these typically rely on *both* an interpretation-based and a compilation-based backend at a high implementation cost.

1.1 Intermediate Representation Levels

The intermediate representation is an important design choice for query compilers. Figure 1 illustrates the effect of the IR choice on JIT compile times. Query compilers with high-level IRs, such as C/C++ [16,33,11] or OpenCL and Cuda [9,13,12,29] generally have longer compilation times than query compilers that generate lower-level IRs such as LLVM IR [27,28]. Existing work on JIT compilers, however, shows the feasibility of much shorter compile times [2,8] than those of LLVM. In fact non-database JIT compilers reach break-even points for dynamic compilation versus static compilation already for thousands of records [2]. By contrast, state-of-the-art LLVM-based query compilers have compilation times of tens of milliseconds [27], which is sufficient time to process queries on millions of tuples [7]. This raises the question illustrated by the bar '**?**' in Figure 1: How can such short compilation times be adopted for database systems that perform query compilation?

LLVM IR is general purpose and was designed to serve as backend for the translation of high-level language features [20]. Being general purpose, LLVM is relatively heavyweight and devises a translation stack that is "overkill" for relational workloads. The code for relational queries typically consists of tight loops with conditional code mainly to drop non-qualifying tuples. This plain structure offers potential for much simpler translation techniques than those used by general purpose translators, which leverage complex code analysis and register allocation algorithms.

1.2 Contributions

In this work, we present the intermediate representation *Flounder IR* and the *ReSQL* database system, which represent a new approach to query compilation that targets low compilation latencies.

Flounder IR. We propose Flounder IR as a lightweight domain-specific IR that is designed for fast compilation of database workloads. Flounder IR is close to machine assembly and adds just that set of features that is necessary for efficient query compilation: virtual registers and function calls ease the construction of the compiler front-end; database-specific extensions enable efficient pipelining in query plans; more elaborate IR features are intentionally left out to maximize compilation speed. Along with the IR, we show the techniques that are used by the Flounder library for translation of Flounder IR to machine code.

ReSQL. The ReSQL database system was developed as a showcase for low-latency query compilation with Flounder IR[1]. ReSQL provides a full translation stack from SQL to machine code and supports a variety of queries. We discuss the interaction of ReSQL's translation components with Flounder IR and use the system to perform an experimental evaluation on TPC-H benchmark workloads. The analysis shows that our approach to query compilation reduces compilation times while preserving high processing speeds. We show with speedups up to 5.5× over a state-of-the-art LLVM-based query compiler Hyper, that our approach achieves better tradeoffs between compilation and execution time.

1.3 Outline

This work is structured as follows: The next Section 2, illustrates how query compilers use Flounder IR for query translation. Section 3 then details the design of Flounder IR. Section 4 shows the translation of Flounder IR to machine code. Section 5 discusses further improvements and applications of our approach. Section 6 evaluates the approach experimentally and Section 7 discusses future work. Finally, Section 8 wraps-up the article with a summary.

2 Query Translation

Query compilation typically involves one step that translates relational queries to an *intermediate representation* (IR) and another step that translates the IR to machine code. In the following, we give an overview of how both steps are realized for query compilation with our intermediate representation Flounder IR.

2.1 Query Plan to IR

The first translation step traverses the query plan and builds an intermediate representation of the query func-

[1] The source code of ReSQL and the Flounder library will be available at publication time.

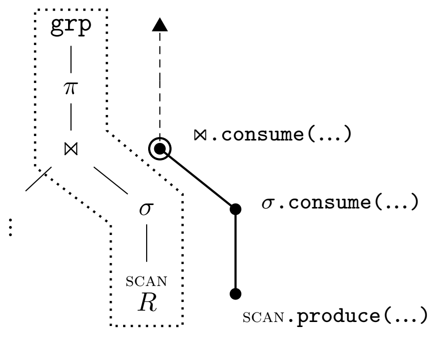


Fig. 2 Translation of the probe-side pipeline of a query plan.

```
TRANSLATE HASH JOIN OPERATOR TO IR
Function ⋈.consume(attributes, caller):
1    if caller is ⋈.left:                                    /* build-side */
2        ht ← createHashtable(...)
3        emit entry ← ht_ins (ht, ⋈.buildKey)  /* get bucket */
4        emit materialize (entry, attributes)   /* write to ht */
5        a_l ← attributes                        /* save build schema */
6    if caller is ⋈.right:                                   /* probe-side */
7        emit entry ← null                                   /* initialize */
8        emit while (true):                  /* loop over join matches */
                 /* probe hash table to get next matching entry */
9            emit entry ← ht_get (ht, ⋈.probeKey, entry)
10           emit  if entry is null:                     /* check result */
11               emit break                              /* no more match */
12           emit dematerialize (entry, a_l)             /* read to regs */
13           ⋈.parent.consume (a_l ∪ attributes, ⋈) /* next ops */
```

Fig. 3 Operator emitter of the hash join operator. We underlined the functionality that is placed in the JIT query.

tionality. A common way to do this is the produce/consume model [27], which emits code for operator functionality either in `produce` or `consume` methods. We call these methods *operator emitters*. Figure 2 illustrates the operator emitters that are executed during translation of a sample query. The *build pipeline* on the left of the join populates the join hash table. It was translated previously. The *probe pipeline*, surrounded by the dotted line, accesses the join hash table. We look at its translation in detail.

The code to scan R was already emitted by the operator emitter SCAN`.produce(...)`. It contains a loop that iterates over the table R and reads its tuples. The code for selection was emitted by σ`.consume(...)` and now the hash join follows with ⋈`.consume(...)`. The implementation of the method is shown in Figure 3, which uses a notation similar to Kersten et al. [15]. Code lines following an **emit** statement are underlined to emphasize that this code is not executed immediately but instead placed in the JIT query. For instance `createHashtable(..)` is not underlined (line 2) and is therefore executed during translation. By contrast `ht_ins(..)` is underlined (line 3) and is therefore placed in the compiled code. This leads to repeated execution of the line for every tuple of the scanned table.

In the example ⋈`.consume(...)` is called from its right child and therefore the probe-side code is produced (lines 7–13). The code first initializes the variable *entry*, which holds hash probe results (line 7) and then loops over the hash join matches (lines 8–13). In the loop, we first call `ht_get(...)` to retrieve the next match (line 9) and then perform a check to exit when no more matches exist (lines 10–11). To process *join matches*, we read the attributes of the match to registers (line 12) and then the join's parent operators place their code by calling `consume(...)` (line 13).

```
 [...] ;child code
 vreg {entry}
 mov {entry}, 0
;loop over matches
loop_headN:
 ;ht_get(..) call
 mcall {entry},{ht_get},
   {ht},{r_a},{entry}
 ;break when entry=NULL
 cmp {entry}, 0
 je loop_footN
 ;dematerialize ht entry
 vreg {s_a}
 vreg {s_b}
 mov {s_a}, [{entry}]
 mov {s_b}, [{entry}+8]
 [...] ;parent.consume(..)
 clear {s_a}
 clear {s_b}
 ;loop foot
 jmp loop_headN
loop_footN:
 clear {entry}
 [...] ;child code
```

Flounder IR
(in-memory)
(a)

```
 [...] ;child code
 mov  r11, 0; init entry
loop_headN: ;while head
 mov  [rsp-8],  r8 ;caller-
 mov  [rsp-16], r9 ;save
 mov  [rsp-24], r10
 mov  rdi, 0x25cac0 ;call
 mov  rsi, r9       ;params
 mov  rdx, r11
 sub  rsp, 24 ;adjust stack
 mov  rax, 0x42fa10
 call rax ;ht_get call
 add  rsp, 24 ;restore stack
 mov  r8,  [rsp-8] ;restore
 mov  r9,  [rsp-16] ;caller-
 mov  r10, [rsp-24] ;save
 mov  r11, rax ;return value
 cmp  r11, 0 ;break condition
 je   loop_footN
 mov  r12, [r11]   ;demate-
 mov  r13, [r11+8] ;rialize
 [...] ;parent.consume(..)
 jmp  loop_headN ;next probe
loop_footN:
 [...] ;child code
```

x86_64 assembly
(in-memory)
(b)

Fig. 4 Intermediate representation of hash join probe functionality (a) and corresponding machine assembly (b).

The resulting intermediate representation is shown in Figure 4 (a)[2]. It performs the described probe functionality. We briefly describe the resulting IR here and

[2] We use an `nasm`-style assembler notation with destination operand on the left and source operand on the right.

provide a detailed description of the used Flounder IR features in Section 3.

The attribute values are held in {r_a}, {s_a}, and {s_b} and the locations of hash table entries in {entry}. The hash_get(...) call is realized with mcall and the loop over the probe matches with a combination of compare (cmp) and two jumps (jmp, je). To read attributes from a hash table entry (dematerialize), we use mov from a memory location in brackets [] to e.g. {s_a}.

2.2 IR to Machine Code

The next step translates the query's intermediate representation to machine code. The machine code needs to follow the application binary interface (ABI) of the execution platform. In this work, we use the target architecture x86_64 [22].

The Flounder IR emitted by the hash join is translated to the machine assembly shown in Figure 4 (b). Several abstractions that were used during IR generation are now replaced by machine-level concepts. E.g. the machine assembly uses processor registers such as r12 instead of {s_a}. Further, the machine assembly uses additional mov instructions to transfer values between registers and the stack, e.g. mov r8,[rsp-8]. The translation process from Flounder IR to machine code needs to manage machine resources such as registers and stack memory and find an efficient way for their use during JIT query execution.

2.3 ReSQL Translation Mechanisms

For a more comprehensive picture, we now describe two more essential translation mechanisms used by ReSQL. First we discuss translation of typed SQL expressions, which are used by operators, e.g. in selection or join criteria. Then we discuss handling of tuples in the implementation of operator emitters.

Expression Translation. To illustrate expression translation, we use the expression 10.0 + 0.34, a sum of two decimal constants, as example. ReSQL uses 64 bit integers for decimal arithmetics and thus represents the values as 100 and 34 along with the *base* and *precision*. The precision is the number of digits in total and base is the number of digits following the decimal point.

For JIT-based evaluation, the expression translator performs two steps. The first step is type resolution, a standard procedure that derives the result type of each expression node. The leaf types decimal(3,1) and decimal(3,2) are given by the constants. The expression translator applies type rules to derive the typed

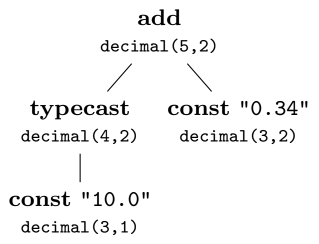


Fig. 5 Typed expression tree for the expression 10.0 + 0.34.

```
;const "0.34"
vreg   {dec_const0}
mov    {dec_const0}, 34
;const "10.0"
vreg   {dec_const1}
mov    {dec_const1}, 100
;typecast [decimal(3,1) to decimal(4.2)]
vreg   {cast_res0}
mov    {cast_res0},  {dec_const1}
clear  {dec_const1}
imul   {cast_res0},  10
;add
vreg   {add_res0}
mov    {add_res0},   {dec_const0}
clear  {dec_const0}
add    {add_res0},   {cast_res0}
clear  {cast_res0}
;[...] work with add_res0
clear  {add_res0}
```

Fig. 6 Flounder IR produced for the expression 10.0 + 0.34.

expression tree shown in Figure 5. One typecast was inserted to maintain the same base for the add. Then the second step emits Flounder IR for the expression tree. Starting with the leaf expressions, code for the evaluation of each node is emitted. The resulting Flounder IR to evaluate the expression is shown in Figure 6. The code uses, e.g. vreg {x} and clear {x} to indicate the validity range of {x}. First both constants are loaded. Then the typecast for {dec_const1} is evaluated by multiplying with 10. Finally the add is evaluated and the result is stored in {add_res0}. The IR-code is inserted into the code frame of the query and translated to machine code along with the query.

Handling of Tuples. In JIT-based execution, the individual values of a tuple are distributed across registers. For the implementation of operator emitters, however, it is still useful to handle tuples as a single entity [15]. ReSQL provides several code generation functions in the Values namespace for this purpose. These are shown in Figure 7. To evaluate the projection expressions from a select-clause, for example, we use tup=Values::evaluate(projs). The result tup is a

`tup = Values::evaluate(expr);`
Evaluate the list of expressions `expr`.

`tup = Values::dematerialize(loc, schm);`
Scan a tuple with schema `schm` from location `loc`.

`hash = Values::hash (tup);`
Hash the tuple `tup`.

`flag = Values::checkEquality(tup1, tup2);`
Check tuples `tup1` and `tup2` for equality.

`Values::materialize(tup, loc);`
Write tuple `tup` to location `loc`.

Fig. 7 Tuple-based code generation methods allow us to handle lists of attribute registers as if they were coherent tuples.

list of virtual registers that hold the expression results, ultimately a tuple. Similarly, lists of virtual registers are used to hold tuples after scanning them or when applying a hash function.

3 Lightweight Abstractions

Flounder IR is similar to `x86_64` assembly, but it adds several *lightweight abstractions*. The abstractions are designed with the interface to the query compiler *and* with the resulting machine code in mind. In this way, Flounder IR passes just the right set of information into the compilation process. For operator emitters, the IR provides independence of machine-level concepts, which allows similar code generation as is typically performed with LLVM. For translation to machine code, the abstractions are sufficiently lightweight to avoid the use of compute-intensive algorithms. Additionally, the IR contains information about the relational workload that enables efficient tuning of the machine code.

In the following, we present the lightweight abstractions. They add several pseudo-instructions, i.e. `vreg`, `clear`, and `mcall` to `x86_64` assembly and use additional tokens, which are shown in braces, e.g. {`param1`}.

3.1 Virtual Registers

An unbounded number of *virtual registers* is a common abstraction in compilers [4]. Query compilers use them to handle attributes without the restrictions of machine registers. When replacing virtual registers with machine registers for execution, general purpose compilers perform live-range analysis [1]. This is rather expensive because compilers consider all execution-paths that lead to a register usage.

Query workloads use virtual registers in a much simpler way than general purpose code. They hold attribute data within a pipeline and the pipeline's execution path only consists of tight loops. This allows query compilers to use a simpler approach that skips live-range analysis. In Flounder IR, operator emitters mark the validity range of virtual registers. The `vreg` pseudo-instruction marks the start of a virtual register usage, e.g. by using

```
;start virtual register use is
vreg {vreg_nameN}
```

and the `clear` pseudo-instruction marks the end of the usage, e.g. with

```
;finish virtual register use
clear {vreg_nameN} .
```

We use these markers in a way similar to scopes in higher-level languages. For instance the Flounder IR in Figure 4 (a) marks the range of the probe attributes {`s_a`} and {`s_b`} to reach around the operators that are contained in the probe loop.

3.2 Function Calls

Being able to access pre-compiled functionality is important for query compilers. It reduces compile times and avoids the implementation cost of code generation for every SQL feature. To this end Flounder IR provides the `mcall` pseudo-instructions to specify function calls in a simple way. For instance

```
;function call to ht_ins
mcall {res} {ht_ins} {param1} ... {paramN}
```

represents a function call to `ht_ins(...)` with parameters `param1` to `paramN` and the return value is stored in {`res`}. A pointer to the function code is provided as an address constant via {`ht_ins`}. This pseudo-instruction is later replaced with an instruction sequence that realizes the calling convention.

3.3 Constant Loads

Large constants, e.g. 64 bit, can not be used as *immediate operands* (`imm`) on current architectures. To use large constants, they have to be placed in machine registers. The *constant load* abstraction in Flounder IR, allows using such constants without restrictions. E.g.

```
;load from 64 bit address with offset
mov {attr} [{0x7fff5a8e39d8} + {offs}]
```

loads data from the address {`0x7fff5a8e39d8`}+{`offs`} to the virtual register {`attr`}. During translation to machine assembly, the address constant will be placed in a machine register.

3.4 Transparent High-Level Constructs

We use *transparent high-level constructs* that mimic high-level language features such as loops and conditional clauses. They are used to generate Flounder IR in operator emitters. For example operator emitters can generate a while loop with the condition {tid} < {len} by using the methods `While(...)`, `close(...)`, and `isSmaller(...)` as shown below.

```
// Produce code for while loop (C++)
wl = While(isSmaller(tid,len)); {
    [...]
} wl.close();
```

This generates the Flounder IR code shown in the margin, that realizes the loop functionality. The start of the loop is marked with the label `loop_headN`. The `cmp` instruction then evaluates the loop condition and `jge` jumps to the `loop_footN`-label at the loop end, if the condition evaluates to false. Otherwise, the loop body is executed and after it, the loop starts over by executing the jump instruction `jmp loop_headN`, which redirects control flow to the loop head.

```
loop_headN:
 cmp {tid},{len}
 jge loop_footN
 ;loop body
 [...]
 jmp loop_headN;
loop_footN:
 ;after loop
 [...]
```

4 Machine Code Translation

In the translation from Flounder IR to x86_64 machine code, the abstractions that facilitated code generation in the previous step are now replaced with machine concepts. A key challenge here is to replace virtual registers with machine registers and to manage spill memory locations for cases of insufficient registers. Finding optimal register allocations is an NP-hard problem and even the computation of approximations is expensive [10]. In the context of JIT compilers, linear scan has been proposed as a faster algorithm [30] and was adopted by LLVM. However, linear scan register allocation is still relatively expensive due to live range computations and increasing numbers of registers.

The following presents a much simpler technique that benefits from the explicit usage ranges marked in Flounder IR. We first show the machine register configuration used by the translator and then show the algorithm for translation of the lightweight abstractions.

4.1 Register Layout

We use a specific register layout for the machine code generated from Flounder IR. The layout is shown in Figure 8. We split the 16 integer registers of the x86_64 architecture into three categories.

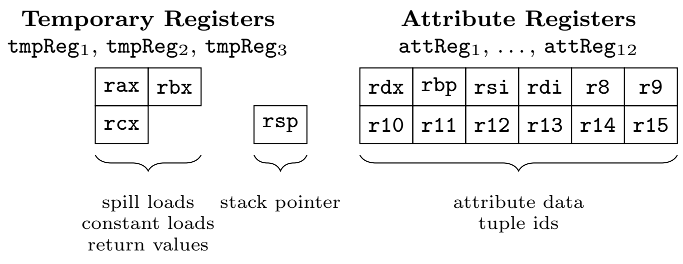


Fig. 8 Usage of machine registers by the translator.

```
TRANSLATE FLOUNDER IR TO MACHINE ASSEMBLY
1  $a \leftarrow 0$                                   /* attribute registers in use */
2  foreach instruction $i$ in input:
3      $t \leftarrow 0$                               /* temporary registers in use */
4      if $i$ is vreg $\{v\}$:                        /* allocate pseudo-instruction */
5          if $a <$ number attribute registers:
6              allocate free $\texttt{attReg}_k$               /* machine register */
7              $a \leftarrow a + 1$
8          else allocate spill location                        /* spill */
9      elseif $i$ is clear $\{v\}$:               /* deallocate pseudo-instruction */
10         if any $\texttt{attReg}_k$ holds $v$:
11             release $\texttt{attReg}_k$                    /* free machine reg */
12             $a \leftarrow a - 1$
13     elseif $i$ is mcall (...):               /* function call pseudo-instr. */
14         emit call-convention code
15     else:                                             /* other instructions */
16         foreach virtual register operand $v$ in $i$:
17             if $v$ is spilled:
18                 emit spill code for $v$ to $\texttt{tmpReg}_t$       /* spilled */
19                 replace $v$ with $\texttt{tmpReg}_t$
20                 $t \leftarrow t + 1$
21             else replace $v$ with $\texttt{attReg}_k$   /* machine register */
22         foreach constant load operand $c$ in $i$:
23             emit load $c$ to $\texttt{tmpReg}_t$      /* place c in temp reg */
24             replace $c$ with $\texttt{tmpReg}_t$ in $i$
25             $t \leftarrow t + 1$
26         emit $i$                              /* output native instruction */
```

Fig. 9 Pseudocode for the translation of Flounder IR to machine assembly. The code is translated in one pass.

We use twelve attribute registers $\texttt{attReg}_1, \ldots, \texttt{attReg}_{12}$ to carry attribute data and tuple ids. We use three temporary registers `tmpReg1`, `tmpReg2` and `tmpReg3`, which are-multi purpose for accessing spill registers and constant loads. Lastly, we use the stack pointer `rsp` to store the stack offset. The stack base pointer `rbp` is repurposed for attribute data and not used for the stack.

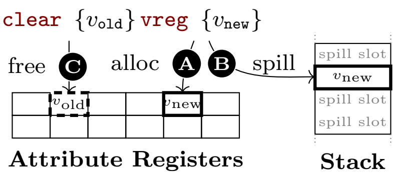
4.2 Translation Algorithm

The translation algorithm translates Flounder IR to x86_64 assembly in one sequential pass over the code.

It replaces the Flounder abstractions with machine instructions, machine registers, and stack access. The algorithm is shown in Figure 9.

When iterating over the IR elements, the algorithm keeps track of a, the number of in-use attribute registers (line 1), and t, the number of temporary registers per instruction (line 3). We describe the translation in three parts. The first part is register allocation, then the replacement of virtual operands with machine operands in instructions, and finally function calls.

Register Allocation. Register allocation is used to decide which virtual registers are stored in machine registers and which virtual registers are stored on the stack. Register allocation does not produce code directly, but it sets the allocation state for spill code and operand replacement. The procedure is illustrated below.



When a `vreg` {$v_{\mathtt{new}}$} pseudo-instruction is encountered (line 4), there are two options. In case Ⓐ there are sufficient machine registers available and we assign one of them to $v_{\mathtt{new}}$ (lines 5-7). In case Ⓑ all machine registers are occupied and we assign a spill slot on the stack (line 8). For `vreg` {$v_{\mathtt{old}}$}, illustrated by Ⓒ, any machine registers assigned to $v_{\mathtt{old}}$ are freed (line 11).

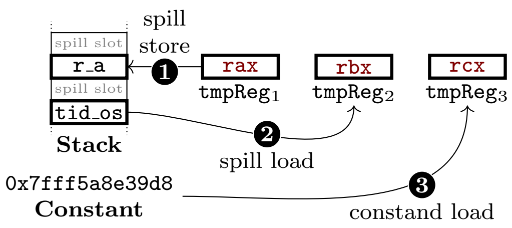
This assignment procedure has the effect that spilled virtual registers remain spilled. However, this happens only when the pipeline requires to hold more than 12 attributes simultaneously. As query compilers typically choose pipeline boundaries such that the data volume per tuple fits into the processor registers, this technique is a perfect match for query compilation.

Spill Code and Operand Replacement. For each instruction, operands that use *constant loads* or *virtual registers* have to be replaced with machine-compatible operands. Virtual registers that were assigned with machine registers are simply swapped (line 21). For the other cases, the algorithm uses `tmpReg`$_1$ to `tmpReg`$_3$ to hold values temporarily per instruction. Three registers are sufficient for this purpose as this is the highest numner of non-immediate operands per instruction. As an example, we look at the following instruction.

```
mov {r_a}, [{0x7fff5a8e39d8}+{tid_os}]
```

It reads an 8 byte value with the offset {`tid_os`} from the memory address `0x7f...` and stores it in {`r_a`}. The address is too large for an immediate operand and we assume for illustration purposes that both virtual registers {`r_a`} and {`tid_os`} are spilled.

The translator assigns temporary registers to each operand and emits *spill code* that exchanges values between spill slots and temporary registers. This is performed in pseudocode lines 16–26 and illustrated below.



The algorithm enumerates the virtual register accesses (lines 16-21) and the constant loads (lines 22-25) from the instruction. It assigns one of the temporary registers tmpReg$_1$ to tmpReg$_3$ to each. In step ❶ the translator assigns `tmpReg`$_1$ (`rax`) to the operand {`r_a`}. This is the only output operand of the instruction and the operator emits a store to {`r_a`}'s spill slot on the stack. Step ❷ assigns `tmpReg`$_2$ (`rbx`) to the operand {`tid_os`}. The translator emits a load to retrieve the value from its spill slot. Step ❸ assigns `tmpReg`$_3$ (`rcx`) to the constant load of address `0x7f...` . The translator emits a load for the constant. This results in the following machine code sequence, which includes the original `mov` instruction with replaced operands.

```
mov rbx, [rsp-24]         ;load spill tid_os
mov rcx, 0x7fff5a8e39d8   ;load constant
mov rax, [rcx+rbx]        ;instruction
mov [rsp-8], rax          ;store spill r_a
```

Calling Conventions. During translation the `mcall` IR-instruction is replaced with a machine code sequence that performs the function call. To this end, a *calling convention* is applied, which specifies rules for the execution of function calls on a given hardware platform. It specifies the way registers are preserved across the call, how parameters are passed, and how the stack frame is adjusted. For the `x86_64` calling convention, the calling function preserves up to 7 integer registers (*caller-save registers*) and passes up to 6 parameters in integer registers before using the stack for parameter passing [22].

The call translation is initiated in line 14 of the Flounder IR translation algorithm (Figure 9). The machine register allocation to the point of the call is known. This allows us to generate a call sequence that is tailored to the current register usage.

The `mcall` translation algorithm is specified in Figure 10 and explained in the following. We use the call to `ht_get(..)` from a previous example (Figure 4).

```
mcall {entry}, {ht_get}, {ht}, {r_a}, {entry}
```

```
TRANSLATE mcall ret, func, p_0, …, p_n
 1 foreach $p$ in {ret, p_0,…,p_n}: /* replace virtual registers */
 2     if $p$ is virtual register:      /* and use machine operands */
 3         replace $p$ with attribute register or stack location
 4 $R_{\text{caller-save}}$ = {rsi, rdi, r8, r9, r10, r11}  /* A caller-save */
 5 foreach register $r$ in $R_{\text{caller-save}}$
 6     if $r$ is allocated:                         /* check use */
 7         emit save $r$ to stack
 8 $R_{\text{param}}$ = {rdi, rsi, rdx, rcx, r8, r9}  /* B set parameters */
 9 foreach parameter $p_i$ in $p_0$, …, $p_n$:
10     src ← $p_i$
11     if $p_i$ was overwritten:              /* handle overwrites */
12         src ← stack backup of $p_i$
13     emit mov $R_{\text{param}_i}$, src
14 stackOffset ← total stack usage       /* C boilerplate call */
15 emit sub rsp, stackOffset
16 emit mov rax, func
17 emit call rax
18 emit add rsp, stackOffset
19 foreach register $r$ in $R_{\text{caller-save}}$ /* D restore caller-save */:
20     if $r$ is allocated
21         emit restore $r$ from stack
22 emit mov ret, rax                   /* get return value (C) */
```

Fig. 10 Translate `mcall` IR-instruction to machine code that realizes the `x86_64` call-convention.

It has the return value {`entry`}, the function address {`ht_get`}, and the parameters {`ht`}, {`r_a`} and {`entry`}. To derive the call-convention instruction sequence, the translator first replaces these operands with the already allocated machine operands (lines 1–3).

```
mcall r11, 0x42fa10, 0x25cac0, r9, r11
```

Then the translator generates code that performs the following four steps:

- **A** Save caller-save registers that are in-use on the stack. These are `r8`, `r9`, `r10` in the example (lines 4-6).
- **B** Assign parameter registers in the order specified by the ABI (lines 7-12). We assign `0x25cac0` to `rdi`, `r9` to `rsi`, and `r11` to `rdx`.
- **C** Place boiler-plate code to modify the stack frame, jump into the function, and to retrieve the return value (lines 13-17,21).
- **D** Restore caller-save registers (lines 18-20).

This results in the instruction sequence shown in Figure 11 that realizes the call in machine assembly. The instructions are annotated with **A** to **D** to indicate the step that generated them.

5 Getting More Out of Flounder

Flounder IR is a near-hardware representation for database processing functionality. This property enables additional uses and benefits for the IR. We present ideas

```
mov  [rsp-8],  r8    ;A save caller-save
mov  [rsp-16], r9
mov  [rsp-24], r10
mov  rdi, 0x25cac0   ;B assign parameters
mov  rsi, r9
mov  rdx, r11
sub  rsp, 24         ;C boilerplate call
mov  rax, 0x42fa10
call rax
add  rsp, 24
mov  r8,  [rsp-8]    ;D caller-save restore
mov  r9,  [rsp-16]
mov  r10, [rsp-24]
mov  r11, rax        ;(C get return value)
```

Fig. 11 Instruction sequence for the example function call.

on taking the IR's database specialization further by adding additional domain knowledge to the language. Then we show prefetching as an example of utilizing such domain knowledge. Finally, we discuss the use of Flounder IR as compilation vehicle for higher-level IRs.

5.1 Utilizing Additional Database Knowledge

The domain specialization makes Flounder IR receptive to utilizing particular *database knowledge*. This idea can be extended in the way Flounder IR uses types. Currently it only uses machine datatypes. Alternatively, we can add SQL types to the IR. This simplifies the translation from SQL to Flounder because operator translators can directly emit instructions on SQL types. At the same time the responsibility of implementing SQL types and their special type characteristics moves down one level to the IR translation. This may open up interesting new ways for handling `NULL`-logic or types with multi-register representations (e.g. 128 bit decimals). The translator has the opportunity to apply simpler or unified logic to handle such characteristics.

Many database operators have optimized implementations that leverage hardware features, e.g. sort and hash-based operators [3]. Specifically applying vectorization techniques (e.g. AVX) has proven to be beneficial [36]. Flounder IR is a good match for such techniques because it gives explicit control over the instructions that are used. This helps to clearly express the way hardware optimizations are applied, which can be difficult with high-level languages that abstract hardware details. Similar to passing specific implementation aspects, additional hints about the database or about database statistics may be used. For instance information about relation and tuple sizes can be leveraged by the compiler for loop unrolling and prefetching. Hints about predicate selectivities are beneficial in estimating which branches are likely to be taken.

5.2 Memory Optimizations via Prefetching

Memory *bandwidth* and *latency* are the most limiting resources for in-memory data processing systems [6]. While current hardware handles local and predictable memory access patterns effectively, more unpredictable patterns typically lead to memory stalls, which leave the CPU idle and slow down processing.

As a solution, current hardware provides prefetch instructions, that can be used by developers to place hints about data that is worthwhile to pre-load. Algorithms that leverage this feature [18,23,31], however, are intricate to design and require careful understanding of the hard- and software. Compilers on the other hand, which insert prefetch instructions automatically (LLVM [24], GCC [21]), need to perform extensive analysis of the program's memory access patterns

To simplify prefetching, Mühlig et al. show Mx-Tasks [26], which annotate tasks (small program pieces) with domain knowledge about the required data. Similarly Flounder can leverage such information coming from the query compiler to benefit from prefetching without interfering with its compile time goal. As a poster case, we built a *scan prefetcher* that inserts prefetches for tuples that are read from memory. Since only complete cache lines can be addressed by the prefetch instructions, the scan prefetcher unrolls the loop to process a full cache line per iteration. The optimization is applied before machine code translation with a small overhead in compilation time.

Scan Prefetcher. To illustrate how the scan prefetcher works, we look at the following code. The code initializes {`scanLoc`} with the relation address {`rel`} and iterates over the relation's 16 byte tuples.

```
 mov {scanLoc}, {rel}
scan_loop_head:
 [...] ;check condition
 [...] ;loop body
 add {scanLoc}, 16
 jmp scan_loop_head
scan_loop_foot:
 [...]
```

When dealing with tuples (in row-based systems) or column-widths (in column-based systems) smaller than a cache line, adding a single prefetch at the start of the loop is insufficient. This results in unnecessary costs for the execution of prefetch instructions because each prefetch handles a *full cacheline*. To address this, the scan prefetcher unrolls the scan loop. In our case a cache line (64 bytes) contains four tuples (4×16 bytes) and the loop is unrolled four times. The following code includes the unrolled loop and prefetching:

```
 mov {scanBase}, {rel}
scan_loop_head:
 [...] ;check condition
 ;prefetch tuples {i+4,i+5,i+6,i+7}
 prefetch [{scanBase}+64]
 [...] ;loop body iteration i
 [...] ;loop body iteration i+1
 [...] ;loop body iteration i+2
 [...] ;loop body iteration i+3
 add {scanBase}, 64
 jmp scan_loop_head
scan_loop_foot:
 [...] ;handle <4 remaining iterations
```

The unrolled loop uses {`scanBase`} to iterate over the relation in steps of four tuples. After checking the loop condition, a prefetch for the succeeding iteration is issued. The unrolled loop body executes four iterations, which collectively read one cache line. By matching the loop granularity with the prefetching granularity, efficient prefetching of the scanned tuples is added.

The *prefetch distance*, however, constitutes a significant factor for preloading the data from memory into the cache at the right time. If the data is accessed too early or too late, the prefetch will be inefficient. While we used a distance of 1 in this example to initiate the prefetch of cache line $k + 1$ before processing the tuples located in cache line k, the IR might help to optimize the timing of the prefetch instruction in the future. The appropriate distance depends on the time, respectively the executed instructions, between initiating the prefetch and accessing the data. By exactly knowing which (CPU) instructions are executed within the scan loop, the time between prefetch and actual access becomes predictable to optimize the prefetch distance accordingly. The ideal amount of instructions between prefetch and access, however, depends on the underlying hardware and needs to be ascertained carefully.

5.3 Higher-Level IRs

Other IRs that describe data processing on a higher level than Flounder IR are frequently used. They are used as translation step for a specific query processing paradigm. For instance MonetDB uses MAL [5] for its column-style processing approach and SQLite uses a (high-level) bytecode representation for its bytecode interpreter [34]. Alternatively higher-level IRs can be used as an abstraction layer. As such they enable database systems to target different parallel hardware architectures [9,29] or to handle multiple processing paradigms. The IR Voila [14], for instance, provides a representation that is suitable for compiled and interpreted execution. We take Voila's scatter operation as example to illustrate how Flounder can be leveraged for compiled execution of this IR. The scatter operation is used by hash-based operators to write values to the hash table. For example

```
// Voila scatter operation: Write key to HT
scatter ( ht.k1, new_pos |can_scatter, t[0] )
```

scatters the value `t[0]` to the hash table key location `k1` of the bucket `new_pos`. The scatter is executed conditionally depending on the flag `can_scatter`. Translation to Flounder IR can implemented as a operator emitter, similar to Section 2.1. The Voila operation translates a short sequence of Flounder instructions:

```
;Scatter op in Flounder IR
 cmp {can_scatter}, 0
 je  afterScatter
 mov [{new_pos}+4], {t0}
afterScatter:
```

The `cmp` and `je` instructions evaluate {`can_scatter`} to skip processing if necessary. Then `mov` performs the actual write of {`t0`} to the hash bucket with base address {`new_pos`} and an exemplary offset `+4`.

6 Evaluation

This section evaluates our approach of using a simple IR for query compilation that is specialized to relational workloads over using a general purpose IR. We use the micro prototype of a query compiler to evaluate the characteristics of different IR's along with their translation libraries. Then we use the ReSQL database system that was built on top of Flounder IR to evaluate the real world performance of our approach against other state-of-the-art systems.

Micro Prototype. We use a smaller query compiler prototype that supports translation of query plans to *both* Flounder IR and LLVM IR. This allows us to evaluate the performance of both IRs on the same system. The prototype is used to execute the workloads from Figure 12. Flounder emits the binary representation of compiled queries with the `AsmJit` library [17] to avoid the overhead of running external assemblers, e.g. `nasm`. For LLVM IR, the machine code is generated by the LLVM library's JIT functionality. We use `O0` and `O3` optimization levels for tradeoffs between compilation time and code quality.

```
SELECT AVG(r.e)
  FROM r,s --len(r)=len(s)=l
 WHERE r.b = s.d
   AND r.c BETWEEN 40 AND 50
```

Q📊: Vary relation lengths (l).

```
SELECT r.a1, r.a2, ..., r.ap
  FROM r
 WHERE r.a1 < c
```

$\mathbf{Q}_\pi$: Vary projection complexity (p).

```
SELECT r1.a, r2.a, ..., rj.a
  FROM r1, r2, ..., rj
 WHERE r1.a = r2.a
     ...
   AND rj-1.a = rj.a
```

$\mathbf{Q}_\bowtie$: Vary join complexity (j).

```
SELECT r.a
  FROM r
 WHERE r.a != c1
   AND r.a != c2
     ...
   AND r.a != cs
```

$\mathbf{Q}_\sigma$: Vary selection complexity (s).

Fig. 12 Query templates used to vary query characteristics.

Database Systems. We built the JIT-compiling database system ReSQL, which uses Flounder IR during compilation and has the ability to run various SQL queries. This allows us to evaluate the real world performance by executing TPC-H benchmark queries. For comparison, we use one compilation-based system Hyper [27] and one interpretation-based system DuckDB [32]. We use Hyper version `v0.5-222`, which executes queries by JIT compiling via LLVM. We use DuckDB version `v0.2.5`, which executes queries with vector-at-a-time processing [7] for cache-efficiency. In its current development state, ReSQL only supports single-threaded execution. We configured all systems to run single-threaded for a fair comparison. Furthermore, ReSQL's query planner does not yet support sub-queries. Therefore we only use benchmark queries that do not contain sub-queries.

Design of Characteristic Workloads. We use four query templates that allow us to evaluate different query characteristics. The templates are specified in Figure 12 in an SQL-form that uses additional integer parameters for configuration. The parameter l varies the data size in **Q**📊. Parameters p, j, and s vary query complexity in $\mathbf{Q}_\pi$, $\mathbf{Q}_\bowtie$, and $\mathbf{Q}_\sigma$ respectively. The attribute data is generated from uniform random distributions and the raltions have the following sizes: **Q**📊 has l tuples for r an s, $\mathbf{Q}_\pi$ has 1 M tuples, $\mathbf{Q}_\bowtie$ has 10 K tuples per join relation, and $\mathbf{Q}_\sigma$ has 1 M tuples.

Execution Platform. We use a system with Intel(R) Xeon E5-1607 v2 CPU with 3.00 GHz and 32 GB main memory. The experiments run in one thread. We use operating system Ubuntu 18.04.4 and clang++ 6.0.0 to compile the query compiler and the library for JIT queries. The LLVM backend uses LLVM 6.0.0.

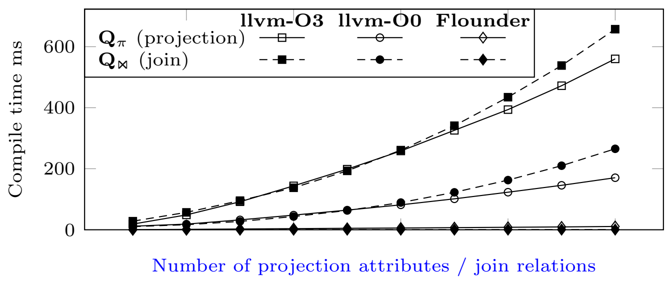


Fig. 13 Effect of query complexity on compilation times for different query compilation techniques.

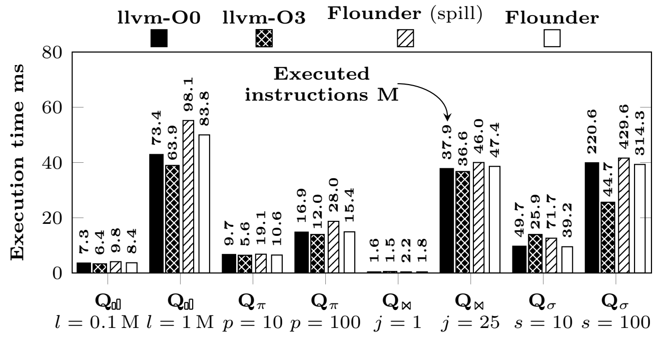


Fig. 14 Time and instruction count for execution of machine code from different query compilation techniques.

6.1 Compilation Times

We compare the machine code compilation times for LLVM and Flounder for $\mathbf{Q}_\pi$ and $\mathbf{Q}_{\bowtie}$. We use $\mathbf{Q}_\pi$ with values of p to project 50 to an extreme case 500 attributes (filter with selectivity 1%). We use $\mathbf{Q}_{\bowtie}$ with values of j to join 2 to 100 relations. We show the results for **Flounder**, **llvm-O0**, and **llvm-O3** in Figure 13.

Observations. For all techniques, the compilation times increase with the query complexity. The compilation times for $\mathbf{Q}_{\bowtie}$ are higher (up to 657 ms) than for $\mathbf{Q}_\pi$(up to 560 ms) and we look in detail at $\mathbf{Q}_{\bowtie}$. With `O0` optimization LLVM has compilation times between 10 ms up to 265 ms. With `O3` compilation times range from 28 ms up to 657 ms. For both levels, the graphs show super-linear growth of compilation times with query complexity. **Flounder** shows lower compilation times that scale linearly between 0.3 ms to 10.8 ms. The highest factor of improvement is 24.6x over **llvm-O0**. and 60.9× over **llvm-O3** (both for 100 join relations). For $\mathbf{Q}_\pi$ **Flounder** has very low compilations times ranging from 0.1 ms (50 attributes) to 0.6 ms (500 attributes). This leads to factors of improvement up to 933× over **llvm-O3**. We attribute this to the time LLVM spends on register allocation. This is due to the large number of virtual registers used for carrying attributes.

6.2 Machine Code Quality

To evaluate machine code quality, we execute two configurations of each query template and measure the *execution time* and the number of *executed instructions*. The results are shown in Figure 14. The bars show the execution time in milliseconds and the number on top shows the executed instructions in millions.

Register Allocation. We analyze the effect of our register allocation strategy on machine code quality. To this end, we look at the techniques **Flounder** (spill) and **Flounder**. The former uses spill access for every virtual register use. The latter allocates machine registers with the translation algorithm. We observe that register allocation reduces the number of executed instruction by factors between 1.2× and 1.8× (with one exception). This shows that our register allocation strategy effectively reduces the amount of executed spill code. We explain the lack of improvement for $\mathbf{Q}_{\bowtie}$ $j = 25$ with a large number of hash table operations, which execute invariant library code. The results show that the register allocation technique reduces execution times for all queries by factors between 1.02× to 1.35×. The factors are not as high as the factors between L1 access and register access. This is because the memory access for reading relation data limits throughput (as is typical for database workloads). The improvements shown by the experiment are due to faster machine register access and execution of less spill code.

Comparison with LLVM. Next we compare the machine code quality of Flounder and LLVM (cf. Figure 14). On average **llvm-O0** executes 1.4× fewer instructions than **Flounder**. The execution times, however, are similar and are longer for **Flounder** only by an average factor of 1.01×. With regard to execution times, the machine code quality resulting from Flounder is similar to **llvm-O0**. We attribute the small time difference despite the higher instruction count to memory bound execution.

The technique **llvm-O3** executes 2.2× fewer instructions than **Flounder** on average. The average factor between the execution times of 1.05× is still low. However, especially queries on larger datasets benefit from the optimizations applied by **llvm-O3**. E.g. the larger variant **Q**📊 1 M executes 1.3× faster. We conclude that despite the much shorter translation times, our compilation strategy produces code with competitive performance to the machine code generated by LLVM.

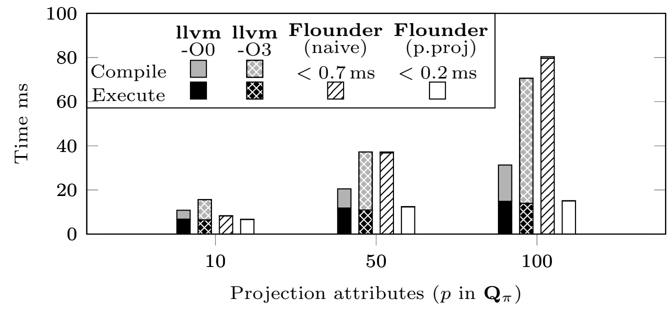


Fig. 15 Processing the projection workload with different compilation and projection techniques.

6.3 Post-Projection Optimizations

The workload $\mathbf{Q}_\pi$ benefits from post-projection optimizations. For increasing numbers of projection attributes p, it is preferable to read attributes a_2 to a_p only for tuples that pass the filter (1% of the relation) instead of performing a full scan. We analyze how the code generation strategies handle post-projection optimization by executing $\mathbf{Q}_\pi$ with $p = \{10, 50, 100\}$. We use the llvm-based techniques, **Flounder** (naive), and **Flounder** (p.proj). The technique **Flounder** (p.proj) produces IR with explicit post-projection; the other techniques produce IR with full scans.

Observations. The experiment results are shown in Figure 15. We observe that **Flounder** (naive) has execution times between 8.2 ms and 79.7 ms, and **Flounder** (p.proj) has lower execution times between 6.6 ms and 15.0 ms. Adding post-projection reduces execution times by factors up to 5.3×. The LLVM-based techniques have execution times between 6.4 ms and 14.8 ms. Despite not using post-projection explicitly, LLVM has similar execution performance as the post-projection strategy. We explain this by LLVM adding a similar optimization during machine code generation.

However, these optimization capabilities of LLVM come at the cost of high compilation times (up to 56.7 ms compared to 0.2 ms for **Flounder**). Although **Flounder** does not apply post-projection optimizations automatically, explicit control over post-projections is preferable for DBMSs, which typically use decision mechanisms for projection strategies.

6.4 Prefetching Optimization

We use $\mathbf{Q}_{\text{📊}}$ and $\mathbf{Q}_\sigma$ to evaluate the effect of applying software-based prefetching. To get an impression for different workload characteristics, we vary the parameters l of $\mathbf{Q}_{\text{📊}}$ and s of $\mathbf{Q}_\sigma$. For both experiments with $\mathbf{Q}_\sigma$, we use a relation length of 5 M tuples, which does not fit into the last-level cache. The experiment uses a different system with Intel (R) Core (TM) i7-9800X CPU with 3.80 GHz. This is because this CPU supports measurement of CPU cycles that were stalled.

Figure 17 shows the execution times and the number of *stalled cycles* during execution. Stalling arises when the CPU waits for data to be loaded from main memory into registers. Ideally, this can be prevented by timely instructing the memory subsystem to prefetch data.

Observations. The results show that the `prefetch` optimization reduces the number of stalled cycles by up to 15 % for $\mathbf{Q}_\sigma$ and up to 6 % for $\mathbf{Q}_{\text{📊}}$. At the same time, execution times reduce up to 11 % ($\mathbf{Q}_{\text{📊}}$) and 14 % ($\mathbf{Q}_\sigma$ with $s = 20$). In particular, for $\mathbf{Q}_{\text{📊}}$, calls of external functions (e.g. building the hash table) interfere with the prefetching. As a result, the stalls are reduced less, compared to $\mathbf{Q}_\sigma$. This may be improved in the future by adapting the prefetching distance. The results also show an increase in compilation time for applying the optimization. However, even with prefetching optimizations, compilation times remain below 0.4 ms, which is sufficiently low to get an overall benefit.

6.5 Overall Performance for Characteristic Workloads

We show a table with the overall performance for each technique in Figure 16. The workloads are the same as in Section 6.2 with two configurations for each template. The relation sizes range from 10 K to 1 M tuples with total attribute numbers between 2 and 100.

Observations. The technique **Flounder** has overall execution times between 0.4 ms and 50.2 ms and **llvm-O0** between 5.3 ms and 74.9 ms. For **llvm-O0**, compilation makes up 46% of the execution on average. For **Flounder** the average is 5%. This leads to better performance of **Flounder** for 7 of 8 queries. For $\mathbf{Q}_{\text{📊}}$ $l = 1\,M$ compilation times are generally low; thus **llvm-O0** achieves a slightly shorter overall time due to 1.15× faster execution. The technique **llvm-O3** has execution times between 11.4 ms and 141.9 ms, which is longer than the other techniques for 7 of 8 queries. The compilation times make up a high percentage of 62% of the overall on average. The highest factor of improvement of **Flounder** over **llvm-O0** is 10.7×. The highest factor of improvement over **llvm-O3** is 23.2×.

6.6 Real World Performance

To evaluate the real world performance of our approach, we execute TPC-H benchmark queries with ReSQL,

		llvm-O0			llvm-O3			Flounder		
		cmpl	exec	**total**	cmpl	exec	**total**	cmpl	exec	**total**
$\mathbf{Q}$▯▯	$l = 0.1\,\mathrm{M}$	4.9	3.5	**8.5**	9.9	3.3	**13.2**	0.1	3.6	**3.8**
$\mathbf{Q}$▯▯	$l = 1\,\mathrm{M}$	4.7	43.6	**48.4**	9.7	38.9	**48.7**	0.1	50.1	**50.2**
$\mathbf{Q}_{\pi}$	$p = 10$	4.0	6.5	**10.6**	9.2	6.4	**15.7**	0.1	6.4	**6.4**
$\mathbf{Q}_{\pi}$	$p = 100$	15.9	14.0	**29.9**	56.7	13.9	**70.7**	0.1	14.0	**14.1**
$\mathbf{Q}_{\bowtie}$	$j = 1$	4.9	0.3	**5.3**	10.9	0.5	**11.4**	0.1	0.3	**0.4**
$\mathbf{Q}_{\bowtie}$	$j = 25$	36.8	38.1	**74.9**	105.2	36.7	**141.9**	2.8	39.1	**42.0**
$\mathbf{Q}_{\sigma}$	$s = 10$	3.8	9.7	**13.5**	7.8	13.9	**21.7**	0.1	9.5	**9.6**
$\mathbf{Q}_{\sigma}$	$s = 100$	10.3	40.0	**50.3**	18.5	25.6	**44.2**	0.2	39.0	**39.2**

Fig. 16 Overall performance for two configurations of each characteristic workloads (values shown are in milliseconds).

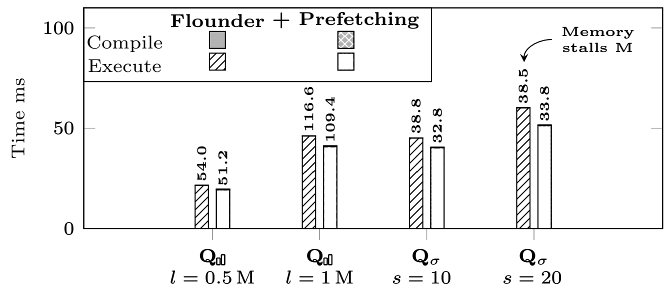
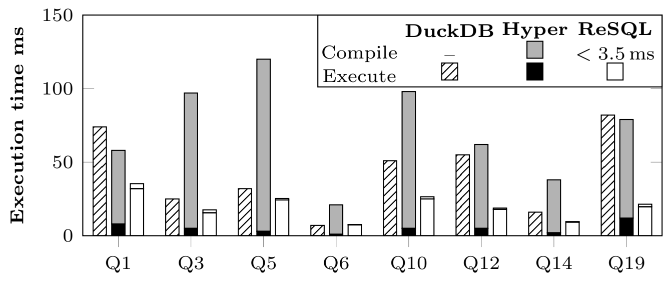
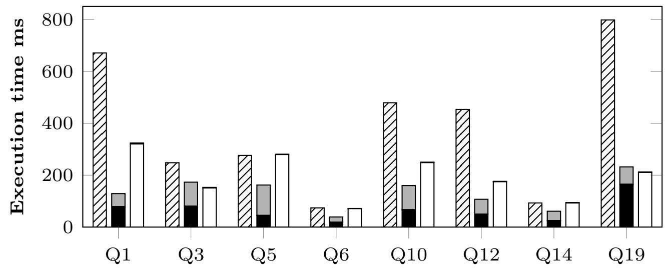


Fig. 17 Effect of using loop unrolling and prefetching for different workloads.



(a) 100 MB database



(b) 1 GB database

Fig. 18 Executing TPC-H queries with DuckDB (interpretation), Hyper and ReSQL (compilation) for two database sizes.

Hyper, and DuckDB. The relative benefit of lowering compilation latencies depends on the size of the processed data. To this end, we evaluate a smaller database with scale factor 100 MB and another database with scale factor 1 GB. We execute those TPC-H queries that are compatible with ReSQL and report compilation times and execution times. We do not show compilation times for DuckDB as it is an interpretation-based engine. The experiment results are shown in Figure 18 **(a)** for the 100 MB database and in Figure 18 **(b)** for the 1 GB database.

Observations 100 MB Database. Excluding compilation times, the JIT-based engines have shorter execution times than the interpretation-based engine DuckDB. DuckDB's execution times range from 7 ms to 82 ms. Hyper's execution times are shorter by 8.3× on average (1 ms to 12 ms). ReSQL's execution times are shorter by 2.3× on average (7 ms to 32 ms). Including compilation times, however, Hyper is slower than DuckDB for 6 out of 8 queries. This is because Hyper's compilation with LLVM takes up to 117 ms. ReSQL has much lower compilation times than Hyper by up to 106.3× (Q5). The highest compilation time of ReSQL is only 3 ms. This makes ReSQL's faster than DuckDB (up to 3.8×) for all but one query (near even for Q6) and faster than Hyper for all queries (up to 5.5×).

Observations 1 GB Database. For the larger database, the execution times (excluding compilation) increase compared to the 100 MB database by 8.9× on average for DuckDB, 13.7× for Hyper, and 10.2× for ReSQL. The compilation times, however, remain unchanged and thus now make up a smaller portion of the overall time for the JIT-based systems. This makes Hyper's overall execution faster than ReSQL for 6 out of 8 queries (1.5× faster on average). Compared to DuckDB, however, ReSQL remains faster by the average factor 1.9× (The factor was 2.0× for the 100 MB database). This is because ReSQL's compilation times have such a small contribution to the overall processing time.

The results show that the simple yet fast compilation approach of ReSQL and Flounder leads to a drastic reduction of compilation times. This leads to faster overall execution times for smaller database sizes (e.g. 100 MB) than state-of-the-art systems. The execution times after compilation of both JIT-based systems are lower than those of the vector-at-a-time engine. This shows that Flounder and ReSQL, despite using simpler translation, can leverage the fast processing speeds of the query compilation approach.

7 Future Work

Flounder IR is highly specialized. It is designed specifically for database workloads and for one hardware architecture only. This was a no-compromise decision for simplicity and translation performance. In the future, it will be interesting to see which *generalizations* can be applied, e.g. targeting multiple hardware architectures, without adding substantial translation cost. In the following, we discuss several aspects of generalization.

7.1 Domain-Specific Processing

Previous work has shown the benefits of combining database processing with other domains, such as data science [35,25]. For Flounder IR, the addition of such other domains would make the IR suitable for use in a wider range of applications. One way to tackle this idea would be to split IR code into database specific parts and domain specific parts. This would allow it to apply Flounder's capabilities (e.g. scope-based register allocation) to the database specific code and use other compiler techniques (e.g. LLVM's register allocator) to the domain specific parts. A challenging aspect of this direction is defining a good interface between both parts, which should allow efficient interaction (e.g. sharing registers) and specialized compilation (e.g. separate basic blocks).

7.2 Hardware Architectures

A straightforward way of supporting other hardware architecture targets is to take Flounder's approach and apply it to a new target. While some aspects may differ, it seems reasonable that the key technique of scope-based register allocation is applicable for most forms of target machine code. By rewriting the IR, however, the emitter functions from the query compiler have to be rewritten aswell. A more sustainable IR design should therefore include the abstraction of most machine-specific concepts and then offer translation for multiple targets from the same IR. It remains an open question how much translation cost such capabilities would add. An early prototype for Intel and ARM architectures showed promising results so far with compilation times similar to Flounder's for basic IR programs.

8 Summary

We showed a query compilation technique that includes all machine code generation steps in the query compiler. The technique uses the intermediate representation Flounder IR that enables *simple translation* of query plans to IR and *fast translation* from IR to machine code. While the translation of query plans to IR is similar to existing approaches, the next step, translation to machine code, is much simpler than in existing techniques. Compared to established low-level query compilers, our approach achieves much shorter compilation times with competitive machine code quality.

The ReSQL database system was built on top of Flounder IR and uses the IR in the translation from SQL to machine code. We use ReSQL to showcase the advantages of Flounder's compilation approach and show that the advantages carry over to real world workloads.

Acknowledgements

We would like to thank the anonymous reviewers aswell as Maximilian Berens for their helpful comments and suggestions. This work was supported by the DFG, Collaborative Research Center SFB 876, A2, and DFG Priority Program "Scalable Data Management for Future Hardware" (TE1117/2-1).

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, principles, techniques. Addison wesley **7**(8), 9 (1986)
2. Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., Bershad, B.N.: Fast, effective dynamic compilation. ACM SIGPLAN Notices **31**(5), 149–159 (1996)
3. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: Sort vs. hash revisited. Proceedings of the VLDB Endowment **7**(1), 85–96 (2013)
4. Bharadwaj, J., Chen, W.Y., Chuang, W., Hoflehner, G., Menezes, K., Muthukumar, K., Pierce, J.: The Intel IA-64 compiler code generator. IEEE Micro **20**(5), 44–53 (2000)
5. Boncz, P.A., Kersten, M.L.: MIL primitives for querying a fragmented world. The VLDB Journal **8**(2), 101–119 (1999)
6. Boncz, P.A., Manegold, S., Kersten, M.L., et al.: Database architecture optimized for the new bottleneck: Memory access. In: PVLDB, vol. 99, pp. 54–65 (1999)
7. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-Pipelining Query Execution. In: Cidr, vol. 5, pp. 225–237 (2005)
8. Bonzini, P.: GNU lightning (2013)
9. Breß, S., Köcher, B., Funke, H., Zeuch, S., Rabl, T., Markl, V.: Generating custom code for efficient query execution on heterogeneous processors. The VLDB Journal **27**(6), 797–822 (2018)
10. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer languages **6**(1), 47–57 (1981)

11. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL server's memory-optimized OLTP engine. In: SIGMOD International Conference on Management of Data, pp. 1243–1254. ACM (2013)
12. Funke, H., Breß, S., Noll, S., Markl, V., Teubner, J.: Pipelined Query Processing in Coprocessor Environments. In: SIGMOD International Conference on Management of Data, pp. 1603–1618. ACM (2018)
13. Funke, H., Teubner, J.: Data-parallel query processing on non-uniform data. PVLDB **13**(6) (2020)
14. Gubner, T., Boncz, P.: Charting the design space of query execution using voila. System **1**(Q3), Q6 (2021)
15. Kersten, T., Leis, V., Neumann, T.: Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. VLDB J **30** (2021)
16. Klonatos, Y., Koch, C., Rompf, T., Chafi, H.: Building efficient query engines in a high-level language. PVLDB **7**(10), 853–864 (2014)
17. Kobalicek, P.: Asmjit Library Library (2018). URL https://asmjit.com
18. Koçberber, Y.O., Falsafi, B., Grot, B.: Asynchronous Memory Access Chaining. Proc. VLDB Endow. **9**(4), 252–263 (2015). DOI 10.14778/2856318.2856321
19. Kohn, A., Leis, V., Neumann, T.: Adaptive execution of compiled queries. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 197–208. IEEE (2018)
20. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO International Symposium on Code Generation and Optimization, pp. 75–86. IEEE (2004)
21. Luk, C., Mowry, T.C.: Compiler-Based Prefetching for Recursive Data Structures. In: ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 222–233. ACM Press (1996). DOI 10.1145/237090.237190
22. Matz, M., Hubicka, J., Jaeger, A., Mitchell, M.: System V application binary interface. AMD64 Architecture Processor Supplement, Draft v0 **99** (2013)
23. Menon, P., Pavlo, A., Mowry, T.C.: Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. Proc. VLDB Endow. **11**(1), 1–13 (2017). DOI 10.14778/3151113.3151114
24. Mowry, T.C., Lam, M.S., Gupta, A.: Design and Evaluation of a Compiler Algorithm for Prefetching. In: ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 62–73. ACM Press (1992). DOI 10.1145/143365.143488
25. Müller, I., Marroquín, R., Koutsoukos, D., Wawrzoniak, M., Akhadov, S., Alonso, G.: The collection virtual machine: an abstraction for multi-frontend multi-backend data analysis. In: Proceedings of the 16th International Workshop on Data Management on New Hardware, pp. 1–10 (2020)
26. Mühlig, J., Teubner, J.: MxTasks: How to Make Efficient Synchronization and Prefetching Easy (to appear). In: Accepted to SIGMOD 2021. ACM
27. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB **4**(9), 539–550 (2011)
28. OmniSci Incorporated: OmniSciDB. `https://www.omnisci.com/` (2019). URL https://www.omnisci.com/platform/omniscidb
29. Pirk, H., Moll, O., Zaharia, M., Madden, S.: Voodoo-a vector algebra for portable database performance on modern hardware. PVLDB **9**(14), 1707–1718 (2016)
30. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS) **21**(5), 895–913 (1999)
31. Psaropoulos, G., Legler, T., May, N., Ailamaki, A.: Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. VLDB J. **28**(4), 451–471 (2019). DOI 10.1007/s00778-018-0533-6
32. Raasveldt, M., Mühleisen, H.: DuckDB: an embeddable analytical database. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1981–1984 (2019)
33. Shaikhha, A., Klonatos, Y., Parreaux, L., Brown, L., Dashti, M., Koch, C.: How to architect a query compiler. In: SIGMOD International Conference on Management of Data, pp. 1907–1922. ACM (2016)
34. SQLite3: Sqlite3. `https://www.sqlite.org/` (2021). URL https://www.sqlite.org/
35. Stonebraker, M., Brown, P., Poliakov, A., Raman, S.: The architecture of scidb. In: International Conference on Scientific and Statistical Database Management, pp. 1–16. Springer (2011)
36. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. Proceedings of the VLDB Endowment **2**(1), 385–394 (2009)