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 queries that finish quickly or that have a high complexity.

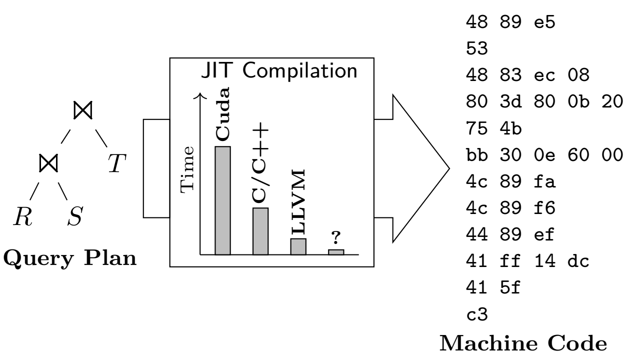
In this work, we present *Flounder IR* and *ReSQL* that represent our new approach to JIT-based query processing with low compilation latencies. Instead of using a general purpose library (e.g. LLVM) for machine code translation, Flounder IR and ReSQL integrate machine-level code generation with the query compiler. This approach enables a much simpler translation process from IR to machine code that can still benefit from the increased processing speeds of the query compilation technique.

The *ReSQL* database system employs a full translation stack from SQL to machine code. This allows us to illustrate the full translation process and to apply our approach to a variety of queries. We analyze the benefit of low-latency query compilation in real world scenarios and show performance improvements up to 5.5× for TPC-H queries 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



```
48 89 e5
53
48 83 ec 08
80 3d 80 0b 20
75 4b
bb 30 0e 60 00
4c 89 fa
4c 89 f6
44 89 ef
41 ff 14 dc
41 5f
c3
```

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 [32]. Approaches to mitigate the impact of compilation time on response time have been proposed previously [22]. However, these typically rely on providing *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++ [19,35,12] or OpenCL and Cuda [10,15,13,31] generally have longer compilation times than query compilers that generate lower-level IRs such as LLVM IR [29,30]. Existing work on JIT compilers, however, shows the feasibility of much shorter compile times [4,9] 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 [4]. By contrast, LLVM-based query compilers have compilation times of tens of milliseconds [29], which is sufficient time to process queries on millions of tuples [8].

LLVM IR is general purpose and was designed to serve as backend for the translation of high-level language features [23]. 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 than performed by general purpose translators, which leverage complex code analysis and register allocation algorithms.

1.2 Contributions

This work presents the intermediate representation *Flounder IR* and the database system *ReSQL*[1], which represent a new approach to query compilation that targets low compilation latencies. This is an extended version of a previously published workshop paper [14]. The extended version adds the ReSQL database system as a real world showcase. The added content includes Section 2.3, which details ReSQL’s translation process, Section 5, which investigates applications of Flounder IR, details on function call translation (in Section 4.2), and an extended experimental evaluation.

Flounder IR. We propose *Flounder IR* as a lightweight intermediate representation for query compilation to reduce compilation times. *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.

ReSQL. The ReSQL database system was developed as a showcase for low latency query compilation with Flounder IR. ReSQL provides a full translation stack from SQL to machine code and supports a variety of queries. In the paper we discuss the design of several of ReSQL’s translation components and we use the system to perform an experimental evaluation on TPC-H benchmark workloads. The analysis shows that our query compilation approach reduces compilation times while preserving improved processing speeds. We show that our approach achieves better tradeoffs between compile time and execution time than previous query compilers.

1.3 Outline

The paper is structured as described in the following. In the next Section 2, we illustrate how ReSQL uses Flounder IR for query translation. Section 3 then details the design of Flounder IR. Section 4 shows the algorithm for translating the IR to machine code. Section 5 discusses further improvements and applications of our translation technique. Then Section 6 evaluates the approach experimentally and finally Section 7 wraps-up the paper with a summary.

2 Query Translation

Query compilation typically involves a 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 Flounder IR.

[1] The source code of ReSQL and the Flounder library will be available at publication time `http://github.com/todo`

```
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
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.

2.1 Query Plan to IR

The first translation step traverses the query plan and builds an intermediate representation of the query functionality. A common way to do this is the produce/consume model [29], 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 the probe-side pipeline of a sample query. The operators of the pipeline are surrounded by a dotted line. In the example, the code to scan R was already emitted by `produce(...)` and for selection (σ) by `consume(...)`. The `consume` call for hash join (⋈) follows next. The code of the hash join operator emitter is shown in Figure 3. The code lines following an **emit** statement are underlined to emphasize that this code is not executed immediately but instead placed in the JIT query.

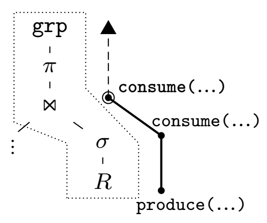


Fig. 2 Query Plan.

In the example, the `consume` method 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
;while head
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 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

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

execution platform. In this work, we use the target architecture x86_64 [25].

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.

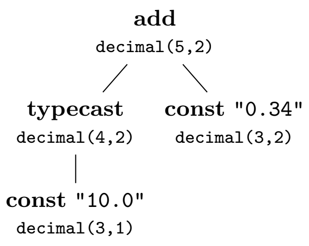
This section provided an overview of query compilation with Flounder IR. The following sections will describe the mechanisms in detail. The next Section 3 shows the abstractions used by Flounder IR during code generation. The subsequent Section 4 will show the translation process from Flounder IR to machine code.

2.3 ReSQL Translation Mechanisms

We just discussed the translation of relational operators. This is one of several translation components used by ReSQL. For a comprehensive picture of the JIT-compiling DBMS, we now describe additional translation mechanisms that are used in the translation from SQL to machine code. We start with the translation of expressions in detail and then continue more briefly with handling of tuples, SQL parsing, relational algebra, and hash-based evaluation of join conditions.

Expression Translation. To illustrate expression evaluation in ReSQL, we use the expression 10.0+0.23, an addition of two decimal constants, as example. An efficient way of executing decimal arithmetic is integer arithmetic along with mechanisms that handle the decimal point, i.e. we encode the constants as integer values 100 and 23 along with the number of decimal digits (*base*). For the addition, we obtain the same base for both summands by multiplying 100 with 10. The integer addition 1000+34=1034 then evaluates to 10.34 by applying the same base.

For JIT-based evaluation, the expression translator of ReSQL performs two steps. The first step is type resolution, which derives the result type for each expression node. Leaf types are given by each constant's number of digits that precede and follow the decimal point. The left constant 10.0 has type decimal(3,1), which means that it has three digits in total and one that follows the decimal point. The right constant 0.34 has type decimal(3,2) respectively. To derive node types the translator applies type rules that are defined for each node class along with its child types. In the process of applying these rules, typecasts are inserted to make operands compatible where necessary. In the example a typecast to decimal(4,2) is added to the left constant for the addition. The resulting expression tree is shown in the margin.



In the second step, the expression translator emits Flounder IR. Starting with the leaf expressions, code for the evaluation of each node is emitted. The resulting Flounder IR below evaluates the full expression from the example. The code uses, e.g. vreg {x}, which has a similar meaning as variable declarations.

```
;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}
```

First the integer representations of both constants are loaded into {dec_const0} and {dec_const1}. Then to evaluate the typecast, {dec_const1} is multiplied with 10 and the result is stored in {cast_res0}. Finally the addition of {dec_const0} and {cast_res0} is performed and the expression result is stored in {add_res0}. Intermediate values are cleared (clear) when they are no longer needed. 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 en-

usage:	`tup = Values::evaluate(expr);`
code:	Evaluate the list of expressions `expr`.
usage:	`tup = Values::dematerialize(loc, schm);`
code:	Scan a tuple with schema `schm` from location `loc`.
usage:	`hash = Values::hash (tup);`
code:	Hash the tuple `tup`.
usage:	`flag = Values::checkEquality(tup1, tup2);`
code:	Check tuples `tup1` and `tup2` for equality.
usage:	`Values::materialize(tup, loc);`
code:	Write tuple `tup` to location `loc`.

Fig. 5 Tuple-based code generation methods. The top line of each pair is a *usage example*. The bottom line describes the functionality of the resulting *generated code*.

tity [18]. ReSQL provides several code generation functions in the `Values` namespace for this purpose. These are shown in Figure 5. To evaluate the projection expressions from a `select`-clause, for example, we use `tup=Values::evaluate(projs)`. The result `tup` is a 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.

SQL Parser. The SQL parser uses a lemon grammar that is translated to C++ code by the lemon parser generator [17]. The grammar uses standard rules, e.g.

```
groupby := GROUP_BY_TK exprList.
groupby := .
```

to define that the `group by`-clause can be either nonexistent or a consist of a `group by`-token with succeeding grouping expressions. The grammer is used to translate SQL queries to the different query entities.

Relational Algebra. After parsing, a set of rules is applied that converts the parsed query into a relational query plan. This plan is lightly optimized by converting to a right-deep tree that uses hash joins for equality conditions (otherwise nested loops) and by pushing selection predicates towards the leaves.

Join Conditions. To implement hash joins, ReSQL uses the pre-compiled hash table operations `ht_get(..)` and `ht_put(..)`. These check only the hash values for equality. For matching hashes, we check the full join conditions via generated code from expression translation.

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, we could design Flounder IR to pass 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 [5]. 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 [2]. 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. using

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

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

```
;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 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
 [...]
```

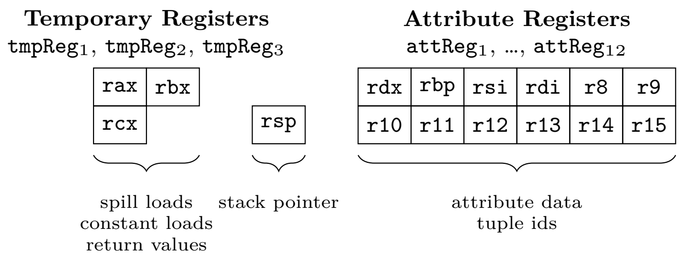


Fig. 6 Usage of machine registers by the translator.

4 Machine Code Translation

This section shows the translation of Flounder IR resulting from plan translation to `x86_64` machine code. The abstractions that were used to facilitate 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 [11]. In the context of JIT compilers, linear scan has been proposed as a faster algorithm [32] and was adopted by LLVM. However, linear scan register allocation is still relatively expensive due to live range computations and increasing numbers of registers.

In this section, we present a much simpler technique that benefits from the explicit usage ranges marked in Flounder IR. In the following, we first show the machine register configuration used by the translator and then we show the algorithm to translate 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 6. We split the 16 integer registers of the `x86_64` architecture into three categories.

We use twelve attribute registers `attReg`$_1$, ..., `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.

```
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 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 attReg$_k$ holds $v$:
11       release 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 tmpReg$_t$     /* spilled */
19         replace $v$ with tmpReg$_t$
20         $t \leftarrow t + 1$
21       else replace $v$ with attReg$_k$   /* machine register */
22     foreach constant load operand $c$ in $i$:
23       emit load $c$ to tmpReg$_t$     /* place c in temp reg */
24       replace $c$ with tmpReg$_t$ in $i$
25       $t \leftarrow t + 1$
26     emit $i$                          /* output native instruction */
```

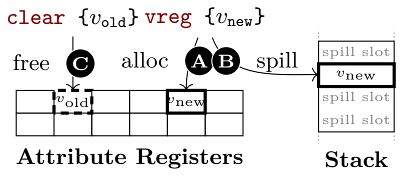
Fig. 7 Pseudocode for the translation of Flounder IR to machine assembly. The code is translated in one pass.

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 7.

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

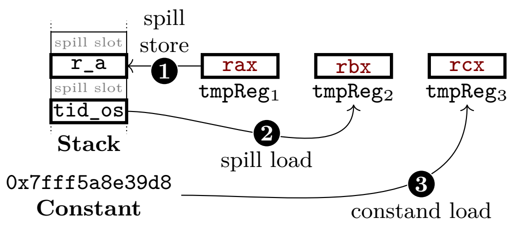
This assignment procedure has the effect that spilled virtual registers remain spilled. However, this happens only when the pipeline requires to hold more than 12 attributes simultaneously. As query compilers typically choose pipeline boundaries such that the data volume per tuple fits into the processor 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 number 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 to 26 and illustrated in the following.



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

```
   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  $R_{caller\text{-}save}$ = {rsi, rdi, r8, r9, r10, r11}   /* A caller-save */
5  foreach register $r$ in $R_{caller\text{-}save}$
6      if $r$ is allocated:                              /* check use */
7          emit save $r$ to stack

8  $R_{param}$ = {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 $R_{param_i}$, 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 $R_{caller\text{-}save}$ /* 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. 8 Translate `mcall` IR-instruction to call-convention code.

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 sets up to 6 parameters in integer registers [25].

The call translation is initiated in line 14 of the Flounder IR translation algorithm (Figure 7). 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 8 and illustrated in the following. We use the call to `ht_get(..)` from a previous example (Figure 4).

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

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 following instruction sequence that realizes the call in machine assembly. The instructions are annotated with steps **A** to **D** that generated them.

```
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)
```

5 Getting More Out of Flounder

Flounder IR provides a near-hardware representation of data processing functionality that enables very fast translation to machine code. These properties enable more uses of Flounder IR than a query compiler target language only. In this section, we first discuss the use of Flounder IR as compilation vehicle for higher-level IRs. Then we discuss opportunities to inject prefetching instructions during translation of Flounder IR to get the most out of the hardware's memory resources.

5.1 Higher-Level IRs

Higher-level IRs are frequently used for data processing workloads. They are used as translation steps to prepare workloads for a specific execution paradigm [6, 36] or as an abstraction layer. As an abstraction layer they enable database systems to target diverse hardware for execution [10,31] or to handle multiple processing paradigms [16].

For instance Voila [16] has a scatter operation specified as `scatter(c,b,x)` that is used by hash-based operators to write values to a 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`. In Flounder IR this Voila operation translates to a short sequence of 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`.

Higher-level IRs such as Voila can be translated to Flounder IR in a straightforward way by implementing a translator. The translator emits the corresponding Flounder IR for each operation of the high-level IR as was illustrated for the Voila scatter operation. By implementing such translators it becomes possible to JIT-compile higher-level IRs for data processing to machine code with very low compilation times.

5.2 Memory Optimizations via Prefetching

Memory *bandwidth* and *latency* are the most limiting resources for in-memory data processing systems [7]. 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 [21,26,33], 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 [27], GCC [24]), need to perform extensive analysis of the program's memory access patterns

To simplify prefetching, Mühlig et al. show MxTasks [28], 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 low compile-times goals. As a poster case, we built a *scan prefetcher* that inserts prefetches for tuples that are read from memory. It 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 code in the margin. The code initializes `{scanLoc}` with the relation address `{rel}` and iterates over the relation's 16 byte tuples. When dealing with tuples (in row-based systems) or column-widths (in column-based systems) that are 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. This results in the following code including prefetching:

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

```
 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 (64 bytes). After checking the loop condition, a prefetch for the succeeding iteration is issued. Then four iterations of the loop body are executed, which collectively read one cache line of tuples. By matching the loop granularity with the prefetching granularity, efficient prefetching for scanned tuples is added.

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 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 9. Flounder emits the binary representation of compiled queries with the `AsmJit` library [20] 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.

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 [29] and one interpretation-based system DuckDB [34]. 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 [8] 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 9 in an SQL-form that uses additional integer parameters. The parameter l varies the data size in $\mathbf{Q}_{▯▯}$. 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 with the following relation sizes: $\mathbf{Q}_{▯▯}$ 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.

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

$\mathbf{Q}_{▯▯}$: 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. 9 Query templates used to vary query characteristics.

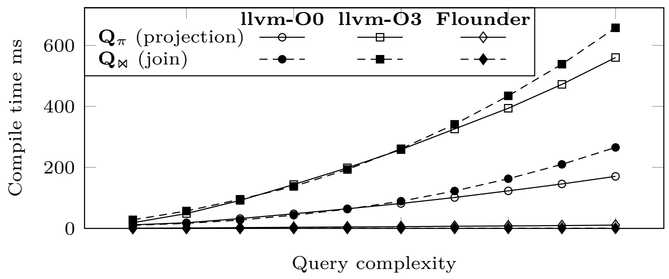


Fig. 10 Effect of query complexity on compilation times for different query compilation techniques.

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.

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 10.

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 60.9x 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 933x 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 to carry attributes for this workload.

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 11. 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 11). 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

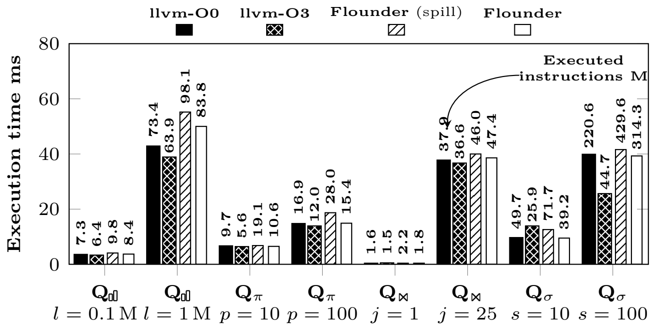


Fig. 11 Time and instruction count for execution of machine code from different query compilation techniques.

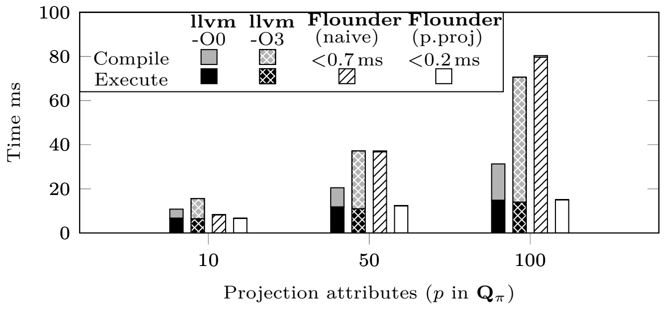


Fig. 12 Processing the projection workload varying compilation and projection techniques.

variant $\mathbf{Q}$📊 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.

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 12. 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 5.3x. The LLVM-based techniques have execution times between 6.4 ms and 14.8 ms. Despite not using post-projection explicitly, LLVM has

		llvm-O0			llvm-O3			Flounder		
		cmpl	exec	**total**	cmpl	exec	**total**	cmpl	exec	**total**
$\mathbf{Q}_{\text{▯▯}}$	$l = 0.1\,\mathrm{M}$	4.9	3.5	**8.5**	9.9	3.3	**13.2**	0.1	3.6	**3.8**
$\mathbf{Q}_{\text{▯▯}}$	$l = 1\,\mathrm{M}$	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. 13 Overall performance for two configurations of each characteristic workloads (values shown are in milliseconds).

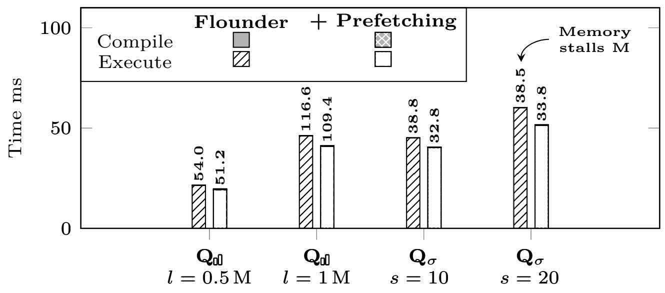


Fig. 14 Effect of using loop unrolling and prefetching for different workloads.

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 14 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 13. 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 10.7x. The highest factor of improvement over **llvm-O3** is 23.2x.

6.6 Real World Performance

To evaluate the real world performance of our approach, we execute TPC-H benchmark queries with ReSQL, 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 and execution times. We do not show compilation times for DuckDB as it is an interpretation-based engine. The results of the experiment are shown in Figure 15 **(a)** (100 MB database) and in Figure 15 **(b)** (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 1.9x (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 Summary

We showed a query compilation technique that includes all machine code generation steps in the query com-

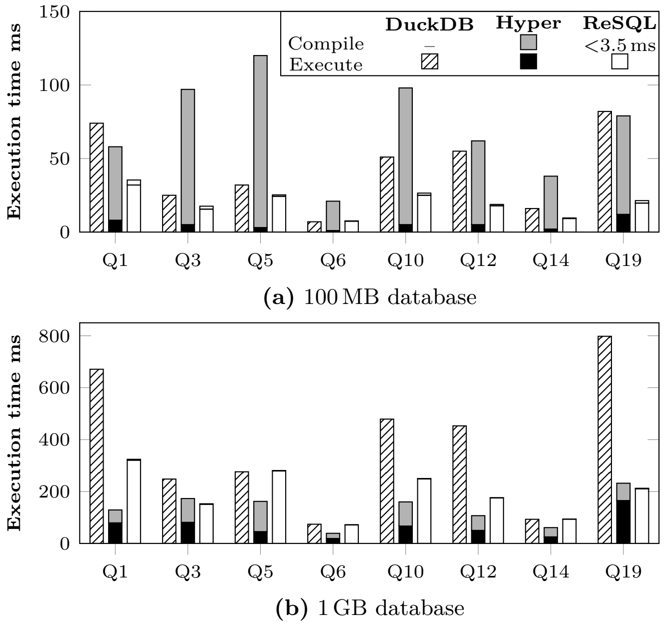


Fig. 15 Executing TPC-H queries with DuckDB (vector-at-a-time), Hyper and ReSQL (JIT compiler) for two database sizes.

piler. 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.

Flounder IR has several applications, which include the query compiler of the database system ReSQL, compilation of other higher-level IRs to machine code, and tuning of machine code for specific hardware architectures. The explicit control over the machine code sequence also makes the approach a good candidate for targeting database-specific architectures [1,3].

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

Acknowledgements

We would like to thank Maximilian Berens and the anonymous reviewers 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. Agrawal, S.R., Idicula, S., Raghavan, A., Vlachos, E., Govindaraju, V., Varadarajan, V., Balkesen, C., Giannikis, G., Roth, C., Agarwal, N., et al.: A many-core architecture for in-memory data processing. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 245–258 (2017)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, principles, techniques. Addison wesley **7**(8), 9 (1986)
3. Arnold, O., Haas, S., Fettweis, G.P., Schlegel, B., Kissinger, T., Karnagel, T., Lehner, W.: Hashi: An application specific instruction set extension for hashing. In: ADMS@ VLDB, pp. 25–33 (2014)
4. Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., Bershad, B.N.: Fast, effective dynamic compilation. ACM SIGPLAN Notices **31**(5), 149–159 (1996)
5. 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)
6. Boncz, P.A., Kersten, M.L.: MIL primitives for querying a fragmented world. The VLDB Journal **8**(2), 101–119 (1999)
7. 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)
8. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-Pipelining Query Execution. In: Cidr, vol. 5, pp. 225–237 (2005)
9. Bonzini, P.: GNU lightning (2013)
10. 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)
11. 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)
12. 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)
13. 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)
14. Funke, H., Mühlig, J., Teubner, J.: Efficient generation of machine code for query compilers. In: Proceedings of the 16th International Workshop on Data Management on New Hardware, pp. 1–7 (2020)
15. Funke, H., Teubner, J.: Data-parallel query processing on non-uniform data. PVLDB **13**(6) (2020)
16. Gubner, T., Boncz, P.: Charting the design space of query execution using voila. System **1**(Q3), Q6 (2021)
17. Hipp, Richard: The lemon parser generator. `https://sqlite.org/src/doc/trunk/doc/lemon.html` (1992). URL https://www.sqlite.org/
18. 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)
19. Klonatos, Y., Koch, C., Rompf, T., Chafi, H.: Building efficient query engines in a high-level language. PVLDB **7**(10), 853–864 (2014)
20. Kobalicek, P.: Asmjit Library Library (2018). URL https://asmjit.com
21. Koçberber, Y.O., Falsafi, B., Grot, B.: Asynchronous Memory Access Chaining. Proc. VLDB Endow. **9**(4), 252–263 (2015). DOI 10.14778/2856318.2856321
22. 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)
23. 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)
24. 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
25. Matz, M., Hubicka, J., Jaeger, A., Mitchell, M.: System V application binary interface. AMD64 Architecture Processor Supplement, Draft v0 **99** (2013)
26. 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
27. 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
28. Mühlig, J., Teubner, J.: MxTasks: How to Make Efficient Synchronization and Prefetching Easy (to appear). In: Accepted to SIGMOD 2021. ACM
29. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB **4**(9), 539–550 (2011)
30. OmniSci Incorporated: OmniSciDB. `https://www.omnisci.com/` (2019). URL https://www.omnisci.com/platform/omniscidb
31. 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)
32. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS) **21**(5), 895–913 (1999)
33. 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
34. Raasveldt, M., Mühleisen, H.: DuckDB: an embeddable analytical database. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1981–1984 (2019)
35. 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)
36. SQLite3: Sqlite3. `https://www.sqlite.org/` (2021). URL https://www.sqlite.org/