Low-Latency Compilation of SQL Queries to Machine Code

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

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

ABSTRACT

Query compilation has proven to be one of the most efficient query processing techniques. Despite its fast processing speed, the additional compilation times of the technique limit its applicability. This is because the approach is most beneficial only when the improvements in processing time clearly exceed the additional compilation time.

Recently the feasibility of query compilers with very low compilation times has been shown. This may prove query compilation as a merely universal approach. In this article and in the corresponding live demo, we show the capabilities of the *ReSQL* database system, which uses the intermediate representation *Flounder IR* to achieve very low compilation times. ReSQL reduces the compilation times from SQL to machine code compared to existing LLVM-based techniques by up to **101.1x** for real-world analytic queries.

PVLDB Reference Format:
Henning Funke and Jens Teubner. Low-Latency Compilation of SQL Queries to Machine Code. PVLDB, 14(12): 2691-2694, 2021.
doi:10.14778/3476311.3476321

1 INTRODUCTION

Query compilation is a processing technique for database queries that achieves very high resource-efficiency and troughput. It uses Just-in-Time (JIT)-compilation to generate custom machine code for every query. This eliminates the overheads of traditional techniques for interpreting query plans and schemas during processing. At the time a query arrives, the query plan and schemas are constants and can be evaluated before processing.

Using compilation, however, introduces additional compilation time, which effectively increases response times. Therefore query compilation is mostly used for applications, where the reduction of execution time clearly exceeds the additional compilation time. This is trivially true for large datasets and even GPU-based techniques with compilation times up to seconds can amortize the cost [2].

Achieving low compilation times, on the other hand, is not only beneficial for the immediate effect of reducing query response times. Furthermore, low compilation times make query compilation more practical. When compilation times are sufficiently low, there is no need for different backends to address different types of workloads. Previous work proposed to *hide* compilation costs by seamlessly switching between interpreted and compiled execution [3]. With sufficiently low compilation times there is no need for the additional implementation cost of redundant backends.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476321

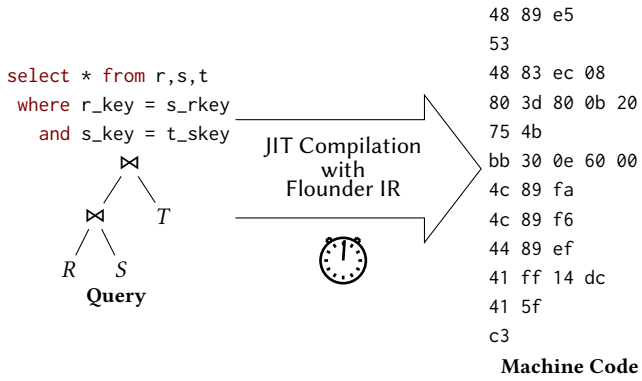


Figure 1: Flounder IR enables compilation of SQL queries to machine code in very short time.

1.1 Low-Latency Query Compilation

Query compilation typically involves two steps. The first translates the query to an *intermediate representation* (IR) and the second translates the IR to *machine code*. Especially during the second step, the choice of IR has a strong effect on compilation times.

Lower-level IRs such as LLVM have been used to achieve low compilation times [4, 5]. Machine code generation with LLVM, however, still takes tens of milliseconds, which is sufficient time to process millions of tuples with traditional non-JIT techniques. The benefit of query compilation with LLVM for smaller data sizes is therefore limited.

Our previous work on Flounder IR [1] has shown that query compilers with much lower compilation times than those of LLVM are feasible. To achieve this, Flounder IR uses a set of features that is tailored to relational workloads and runs only very lightweight algorithms during translation to machine code.

1.2 Contributions

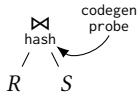
In this work, we demonstrate how Flounder IR is used in a full translation stack from SQL to machine code to enable low-latency JIT compilation of queries. The demonstration enables inspection of different levels of the translation process and shows the practicality of query compilation with Flounder IR. In the following, we first describe IRs for low-latency query compilation (Section 2). Then we present how the ReSQL database system is used for the demonstration (Section 3). Finally Section 4 evaluates the translation performance and Section 5 wraps up with a summary.

2 LOW-LEVEL IRS FOR FAST TRANSLATION

Low-level IRs such as Flounder and LLVM are used to achieve low compilation times during query compilation. Query compilation is

a two-step process that involves first translation of queries to IR and second translation of IR to machine code. For LLVM, the second step is still fairly time-consuming relative to the processing speed of queries. Flounder has the ability perform the translation to machine code in much shorter time. This is achieved by simplifying and tailoring the IR and its translation process to relational workloads.

We now want to point out differences and similarities between both IRs along with an example. To this end, we take the query plan shown in the margin and discuss the translation of the hash join probe operator. We assume that the code for hash join build has already been generated. Now follows the code generation for probing the hash table with tuples from S.



In the following, we first look at a high-level description of the probe funtionality and then we look at both low-level IRs that enable fast compilation. For more details on query compilation with Flounder IR, we refer to the article [1].

2.1 Join Probe

We first describe the functionality generated for the hash join probe with C-code. Although C is not used as the IR here, this is a straightforward way of describing the functionality.

```
[...] /* child code */
int64_t* entry = null;
while ( true ) {
    entry = ht_get ( ht, s_a, entry );
    if ( entry == null ) break;
    int64_t r_a = entry[0];
    int64_t r_b = entry[1];
    [...] /* parent code */
}
[...] /* child code */
```

The probe entry is initialized with `null` and then hash probes are executed via `ht_get(..)`-calls in a loop. When the call returns `null` there are no further matches and we exit the loop. Inside the loop, the attribute values stored at the hash table location `entry` are read into `r_a` and `r_b`. After this the succeeding (parent) operators execute.

2.2 Low-Level Representation

We now show the low-level representations for the hash join probe functionality previously specified as C code. The low-level IRs consist of instructions with a granularity similar to those executed by the processor. However they include several abstractions to facilitate translation. Figure 2 (a) shows the LLVM IR for the hash join probe and Figure 2 (b) the corresponding Flounder IR. Along with the IR code, we exemplify commonalities and differences between both IRs.

Structure. The structure of Flounder IR is simpler than the structure of LLVM. The latter consists of Basic Blocks that start with a label (e.g. `match:`) and end with a jump `br`. This makes the IR a graph with Basic Block-nodes and edges for jumps between them. Flounder IR is linear and consists only of one instruction sequence. The graph-based representations allows compilers to pick a favorable

```
joinProbe:
  ;get previous probe value
  %prev = phi i64* [ null, %scan ],
              [ %ht_get, %match ]
  ;ht_get(..) call
  %ht_get = call i64* %htGetPtr(
                %ht, %s_a, %prev)
  ;break when entry=NULL
  %1 = icmp ne i64* %ht_get, null
  br i1 %1, label %match,
            label %miss
match:
  ;read ht entry
  %addr0 = getelementptr i64,
           i64* %ht_get, i64 0
  %r_a = load i64, i64* %addr0
  %addr1 = getelementptr i64,
           i64* %ht_get, i64 1
  %r_b = load i64, i64* %addr1
  [...] ;parent code
  br label %joinProbe
miss:
  [...]  ;child code
```

LLVM IR
(a)

```
  [...] ;child code
  vreg {entry}
  mov {entry}, 0
loop_headN: ;while(..)
  ;ht_get(..) call
  mcall {entry},{ht_get},
     {ht},{s_a},{entry}
  ;break when entry=NULL
  cmp {entry}, 0
  je loop_footN
  ;read ht entry
  vreg {r_a}
  vreg {r_b}
  mov {r_a}, [{entry}]
  mov {r_b}, [{entry}+8]
  [...] ;parent code
  clear {r_a}
  clear {r_b}
  jmp loop_headN
loop_footN:
  clear {entry}
  [...] ;child code
```

Flounder IR
(b)

Figure 2: Intermediate representation of the hash join probe operator in (a) LLVM IR and (b) Flounder IR.

fall-through path. This is not essential for query workloads because the default path is already identified by the query compiler. The simpler representation of Flounder IR improves translation speed.

Virtual Registers. Both IRs use a logically unlimited set of virtual registers. For LLVM their names start with a percent-sign, e.g. `%prev`. For Flounder virtual register names are contained in braces, e.g. `{entry}`. To reduce translation cost, Flounder uses marker-instructions to indicate usage ranges of virtual registers. Their start is indicated e.g. by `vreg {entry}` their end by `clear {entry}`.

Register Allocation. During translation of IR to machine code, the virtual registers are replaced with machine registers. LLVM applies algorithms, such as live range splitting, to find an allocation that uses machine registers efficiently. For Flounder IR the process is simplified by separating machine registers into *attribute registers* and few *temporary registers*. Attribute registers are allocated for attribute values in a single scan over the IR. Temporary registers are used to access spill-values on the stack and other purposes.

3 DEMONSTRATION: RESQL DBMS

The demonstration is based on the database system ReSQL that was built ontop of Flounder IR. ReSQL allows JIT-based processing of database workloads with very low compilation times. For the demonstation users will be provided with the command line interface for an instance of ReSQL. A screenshot of the command line interface is shown in Figure 3.

```
>select * from region
 Query plan:  { MaterializeOp  { ScanOp(5) } }

Emitted 83 machine instructions.
compile: 0.284119 ms
execute: 0.033035 ms
```

r_regionkey BIGINT	r_name CHAR(25)	r_comment VARCHAR(152)
0	AFRICA	lar deposits. blithely fi..
1	AMERICA	hs use ironic, even reque..
2	ASIA	ges. thinly even pinto be..
3	EUROPE	ly final courts cajole fu..
4	MIDDLE EAST	uickly special accounts c..

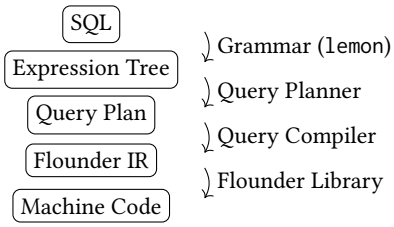
5 tuples

```
>
```

Figure 3: Command line interface of ReSQL.

3.1 Just-in-Time Compilation

The demo system allows users to enter and execute SQL statements live on a sample database. JIT-Compilation is then performed for many SQL queries in very short time. ReSQL translates the queries with the following translation stack:



Translation starts by parsing SQL statements to an expression tree. The query planner transforms the expression tree to a query plan, which is the input for the query compiler. The query compiler then translates each relational operator to Flounder IR, as was shown in Section 2. Finally, the Flounder library translates the IR to binary machine code, which is ready for query execution. The demonstration system uses these steps for all queries and reports compilation and execution times.

3.2 IR Inspection

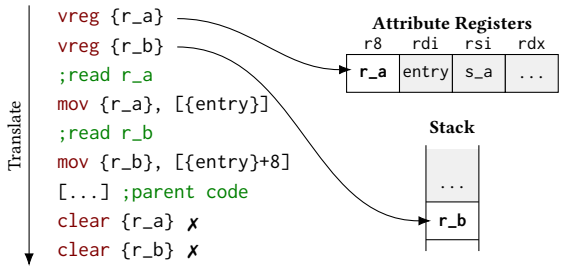
To deep-dive into the translation mechanisms, users can inspect the generated IR code and the resulting machine assembly. This makes it possible to look into various aspects of code generation and to look at the code that was generated for each operator. An example for the IR code of a hash join probe operator is shown in Figure 2 (b).

By looking at the IR code and the machine assembly, we can observe different database and compiler techniques in action. These include register allocation, post-projection optimizations, implementation of relational operators, as well as hash-based aggregations and joins. In the following, we take a look at the dematerialization of hash table entries as an example for IR inspection.

3.3 Example: Register Allocation

To illustrate how the IR inspection allows us to observe register allocation, we get back to the example from Section 2. In the example, hash table entries were read to registers for every match. We now want to inspect the IR and the machine assembly to understand how register allocation is performed for this operation.

The code shown below is an extract from Figure 2 (b) and is responsible for reading the attributes `r_a` and `r_b` from the hash table entry into registers. As scenario for register allocation, we assume that 3 of 4 attribute registers are occupied and only `r8` is still free. No virtual registers were spilled on the stack yet. The attribute registers and the stack are illustrated on the right. During translation, the Flounder translator goes over the instruction sequence and replaces all virtual registers.



Machine Register. When the first `vreg {r_a}` marker-instruction is met, the allocator has to provide a new allocation. As there is a free attribute register `r8` available, it is used for `{r_a}`.

Spill Slot. When the second `vreg {r_b}` marker-instruction is encountered, there are no free attribute registers available. Therefore a stack location is allocated for `{r_b}`.

In the remaining code whenever `{r_a}` is used in an instruction, the virtual register is replaced with `r8`. Whenever `{r_b}` is used, the register allocator emits spill code to exchange its value between the stack and a temporary register. The instruction then operates on the temporary register. At the end of the code both virtual registers are deallocated with `clear` marker-instructions. Thus the register `r8` and `{r_b}`'s stack location are freed. This is indicated by the symbol ✗.

```
mov r8,      [rdi]    ;read r_a
mov rax,     [rdi+8]  ;read r_b
mov [rsp-8], rax      ;spill store
[...]                 ;parent code
```

Machine Assembly. The translation results in the `x86_64` assembly shown above. The code contains the functionality of the two moves from the Flounder IR code and an additional move for stack access. The `vreg` and `clear` marker-instructions are stripped after driving register allocation. With the first `mov` the attribute `r_a` is read into the attribute register `r8`. With the second `mov` the attribute `r_b` is read into the temporary register `rax`. The third `mov` then stores the temporary register's value to `r_b`'s stack spill location `[rsp-8]`.

4 EVALUATION

We evaluate the compilation performance of ReSQL for several workloads. We focus on compilation times as primary showcase of the demonstration. For execution times, we refer to related work [1].

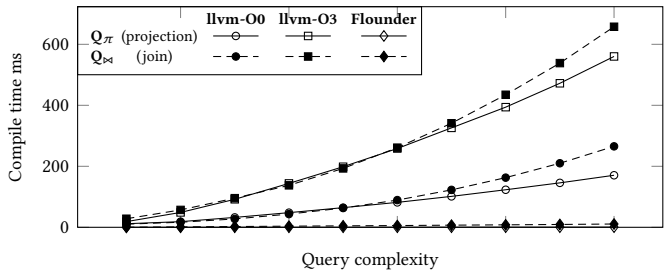


Figure 5: Effect of query complexity on compilation times for different intermediate representations.

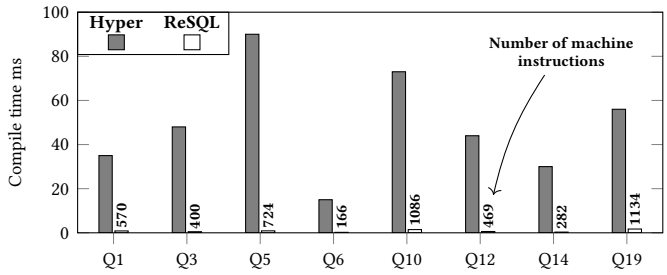


Figure 6: Compile time and number of machine instruction for TPC-H queries with different query compilers.

```
select r.a1, r.a2, ..., r.ap
  from r
 where r.a1 < c
```

Q_π: 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
```

$Q_\bowtie$: Vary join complexity (j).

Figure 4: Complexity query templates.

System. We use a system with Intel(R) Xeon E5-1607 v2 CPU with 3.00 GHz and 32 GB main memory. We use operating system Ubuntu 18.04.4 and LLVM 6.0.0. We use HyPer `v0.5-222-g04766a1` which also uses LLVM for JIT-compilation.

Complexity Queries. We use two query templates Q_π and $Q_\bowtie$, shown in Figure 4 that each have a parameter for query complexity. This allows us to measure the asymptotic compilation times of different JIT techniques when varying the complexity. Q_π is used for queries with varying numbers of projection attributes. $Q_\bowtie$ is used for queries that join varying numbers of relations.

TPC-H Benchmark Queries. We evaluate the compilation times for several TPC-H benchmark queries to characterize the compilation speed for real world analytic queries. We use only queries that contain no sub-queries, because the current query planner of ReSQL does not implement sub-queries yet.

4.1 Asymptotic Compilation Times

We compare the machine code compilation times for LLVM and Flounder for Q_π and $Q_\bowtie$. We use Q_π with values of p to project 50 to an extreme case 500 attributes (filter with selectivity 1%). We use $Q_\bowtie$ with values of j to join 2 to 100 relations. We show the results for **Flounder**, **llvm-O0**, and **llvm-O3** in Figure 5.

For all techniques, the compilation times increase with the query complexity. The compilation times for $Q_\bowtie$ are higher (up to 657 ms) than for Q_π (up to 560 ms) and we look in detail at $Q_\bowtie$. 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 optimization levels, the graphs show super-linear growth of compilation times with query complexity. **Flounder** has lower compilation times that scale linearly between 0.3 ms to 10.8 ms. The highest factor of improvement for $Q_\bowtie$ 24.6x over **llvm-O0**. and 60.9x over **llvm-O3** (both for 100 join relations). For Q_π the highest improvement of **Flounder** over **llvm-O0** is 283x.

4.2 Real World Compilation Times

To evaluate real world compilation times, we execute TPC-H queries with **Hyper** and **ReSQL**. The results of the experiment are shown in Figure 6.

Hyper has compilation times between 15 ms for Q6 and 90 ms for Q5. **ReSQL** has much shorter compilation times between 0.21 ms for Q6 and 1.71 ms for Q19. On average the compilation times of **ReSQL** are **70.1x** shorter than those of **Hyper**. The highest factor of improvement is **101.1x** for Q5. ReSQL has shorter compilation times, because it uses Flounder IR instead of LLVM. Flounder IR is much simpler and tailored to relational workloads, which leads to a significant speed-up of the compilation process.

5 SUMMARY

This article demonstrates the capabilities of Flounder IR as an intermediate representation. We oppose Flounder IR and LLVM IR on a conceptual level and from the viewpoint of compilation performance. The article illustrates the aspects of compilation and query processing that are observable in the live demo.

ACKNOWLEDGMENTS

This work was supported by the DFG, Collaborative Research Center SFB 876, Project A2.

REFERENCES

[1] Henning Funke, Jan Mühlig, and Jens Teubner. 2020. Efficient generation of machine code for query compilers. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–7.
[2] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *PVLDB* 13, 6 (2020).
[3] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 197–208.
[4] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9 (2011), 539–550.
[5] OmniSci Incorporated. 2021. OmniSciDB. https://www.omnisci.com/, last accessed on 07/25/2021.