Compiling Database Queries into Machine Code

Thomas Neumann, Viktor Leis
Technische Universität München
{neumann,leis}@in.tum.de

Abstract

On modern servers the working set of database management systems becomes more and more main memory resident. Slow disk accesses are largely avoided, and thus the in-memory processing speed of databases becomes an important factor. One very attractive approach for fast query processing is just-in-time compilation of incoming queries. By producing machine code at runtime we avoid the overhead of traditional interpretation systems, and by carefully organizing the code around register usage we minimize memory traffic and get excellent performance.

In this paper we show how queries can be brought into a form suitable for efficient translation, and how the underlying code generation can be orchestrated. By carefully abstracting away the necessary plumbing infrastructure we can build a query compiler that is both maintainable and efficient. The effectiveness of the approach is demonstrated by the HyPer system, that uses query compilation as its execution strategy, and that achieves excellent performance.

1 Introduction

Traditionally, database systems process queries by evaluating algebraic expressions. This is typically done using the iterator model [3], in which each operator produces a stream of tuples on demand. To iterate over this tuple stream the *next* function of the operator is called repeatedly. This interface is simple, flexible, easy to implement, and allows to combine arbitrary operators.

The iterator model worked well in the past, as query processing was dominated by I/O, so the overhead of the huge number of *next* calls was negligible. However, with increasing main memory sizes, this situation has changed. CPU and cache efficiency has become very important for good performance. This development has lead to changes in the iterator model in some modern systems. One approach is to produce multiple tuples with each *next* call instead of just one. While such block-wise processing reduces the interpretation overhead, it inhibits pipelining as blocks of tuples are always materialized in order to pass them between operators.

A radically different approach is to avoid the iterator model altogether, and to directly compile queries into machine code. This approach can largely avoid function calls, and can often keep tuples inside CPU registers, which is very beneficial for performance. However, machine code generation is non-trivial, as the algebraic query tree of a query can be arbitrarily complex and it is not obvious how to generate efficient code for it. In this paper we present the compilation strategy of our main-memory database system HyPer [4]. HyPer uses data-centric compilation to compile relational algebra trees to highly efficient machine code using the LLVM

Copyright 2014 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

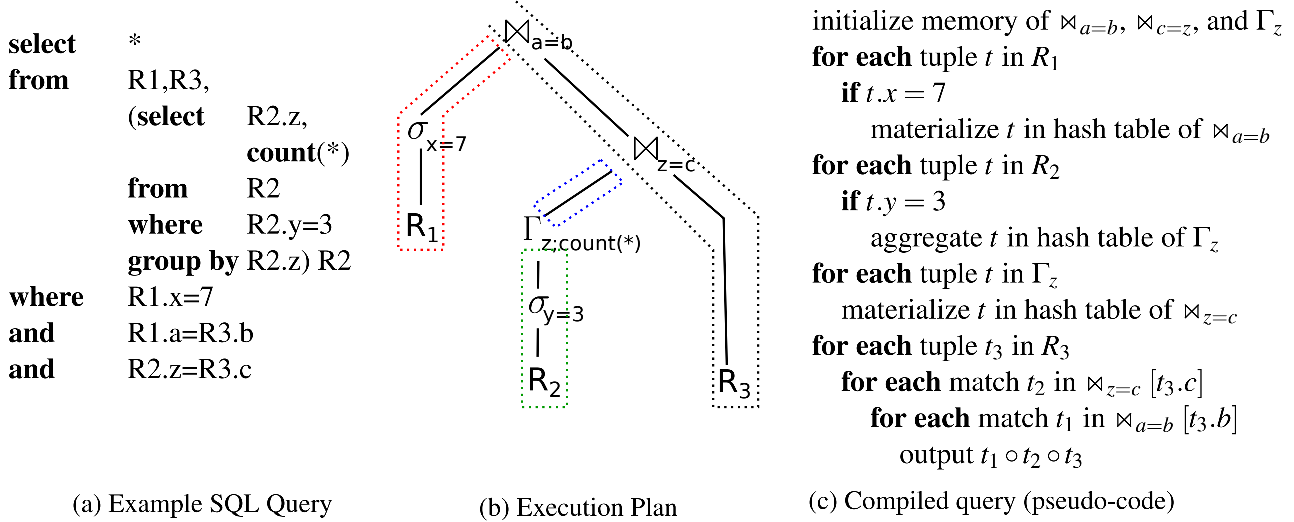


Figure 1: From SQL to executable code

compiler backend. Section 2 introduces the idea of compiling for data-centric query execution. A more detailed description of this approach can be found in [10]. Section 3 introduces a number of building blocks that are used in the query compiler. We evaluate our approach in Section 4. Finally, after discussing related work in Section 5, we conclude in Section 6.

2 Compiling for Data-Centric Query Execution

In most database systems query execution is driven by the algebraic operators that represent the query: The code is organized per-operator, and for each tuple the control flow passes from one operator to the other. The HyPer query engine takes a data-centric point of view: Instead of passing data between operators, the goal is to maximize data locality by keeping attributes in CPU registers as long as possible. To achieve this, we break a query into pipeline fragments, as shown in the execution plan in Figure 1b for the SQL query in Figure 1a. Within each pipeline, a tuple is first loaded from materialized state into the CPU, then passed through all operators that can work on it directly, and finally materialized into the next pipeline breaker. The source operator could be, for example, a table scan that loads the tuples from persistent storage or main memory. Typical intermediate operators that do not materialize are selections or the probe side of an in-memory hash join. These operators perform their work, and potentially filter out tuples, but they do not write to memory. At the end of each pipeline is a pipeline breaker, for example the build side of a hash side that materializes the tuple. The goal of this approach is to access memory as rarely as possible, because memory accesses are expensive. In fact, in a main-memory database system that nearly never accesses disk, we can consider memory as “the new disk”, and want to avoid accessing it as much as possible.

Before execution queries are first processed as usual. After being parsed, the query is translated into an algebra expression, which is then optimized. Since HyPer uses a traditional cost-based query optimizer, our execution strategy does not require fundamental changes to query optimization. Only the cost model needs to be adjusted, mainly by tweaking the appropriate constants. However, instead of translating this expression into physical algebra that is executed (or interpreted), HyPer compiles the algebra expression into an imperative program.

```
scan.produce():
    print "for each tuple in relation"
    scan.parent.consume(attributes,scan)
σ.produce:
    σ.input.produce()
σ.consume(a,s):
    print "if "+σ.condition
    σ.parent.consume(attr,σ)
```

```
⋈.produce():
    ⋈.left.produce()
    ⋈.right.produce()
⋈.consume(a,s):
    if (s==⋈.left)
        print "materialize tuple in hash table"
    else
        print "for each match in hashtable[" +a.joinattr+"]"
        ⋈.parent.consume(a+new attributes)
```

Figure 2: A simple translation scheme to illustrate the *produce/consume* interaction

For the actual execution, we compile each pipeline fragment into one code fragment, as indicated by the matching colors in the execution plan and the pseudo code in Figure 1c. Within each fragment the attributes of the current tuple can be kept in registers, and therefore do not need to be materialized or passed explicitly between the operators. The code for each pipeline stage generally consists of one outer loop that iterates over the input data, before the tuples are further processed within the loop body. To generate such code we must reverse the data flow: Instead of pulling tuples from the input operators as in the iterator model, we *push* tuples towards consuming operators. This process starts by the source operator (e.g., table scan) pushing tuples towards their consuming operator, which continues pushing until the next pipeline breaker is reached. This means that data is always pushed from one pipeline breaker into another. As a result, unnecessary tuple materialization and memory traffic is avoided.

The algebraic operator model is very useful for reasoning about queries during query optimization, but does not mirror how queries are executed at runtime in our model. For example, the three lines in the first (red) code fragment of Figure 1c belong to the table scan R_1, the selection $t.x = 7$, and the hash join $\bowtie_{a=b}$ respectively. Nevertheless, our query compiler operates on an operator tree, which was produced by the query optimizer, and transforms the operator tree into executable code. Conceptually, each operator offers a unified interface that is quite different from the iterator model but almost as simple: It can *produce* tuples on demand, and it can *consume* incoming tuples from a child operator. This conceptual interface allows to generate data-centric code, while keeping the composability of the algebraic operator model. Note, however, that this interface is only a concept used during code generation – it does not exist at runtime. That is, these functions are used to generate the appropriate code for producing and consuming tuples, but they are not called at runtime.

The interface consists of two functions, which are implemented by each operator:

- produce()
- consume(attributes,source)

Logically, the *produce* function asks the operator to produce its result tuples, and to push them towards the consuming operator by calling its *consume* function. Another way to look at this interface is in terms of query compilation: *produce* can be interpreted as generating code that computes the result tuples of an operator, and *consume* generates code for processing one tuple. The code generation model is illustrated in Figure 2, which shows a simplified implementation of the table scan, selection, and hash join operators. One can convince oneself that calling the *produce* function on the root of the operator tree shown in Figure 1b will produce exactly the pseudo-code shown in Figure 1c. It is a bit unusual to have code (here: produce/consume) that generates other code (here: the pseudo-code of the operators). But this is always the case when compiling queries into machine code. The challenge is to make this code generation as painless as possible, without sacrificing the performance of the generated code.

SQL data type	LLVM data types
integer	value: int32, null: int1
decimal(18,2) not null	value: int64
real	value: float64, null: int1
varchar(60)	value: int8*, length: int32, null: int1

(a) Mapping of SQL data types to low-level data types

```
class Value {
   // SQL type
   Type* type;
   // value
   llvm::Value* value;
   // length (optional)
   llvm::Value* length;
   // NULL indicator (if any)
   llvm::Value* null;
}
```

(b) Typed value class

Figure 3: Data type representation

3 Building Blocks

Generating machine code is a very complex task. Fortunately it can be made tractable by using abstraction layers and modularization. A great strength of a compilation-based approach is that with some care these abstraction layers cause nearly no overhead, as they are compile-time abstractions and not runtime abstractions.

On the lowest layer the database needs a mechanism to generate machine code for the target machine. Writing machine code generators by hand is tedious, error prone, and not portable. Therefore, it is highly beneficial to use a compiler backend like LLVM [7] or C-- [11]. In principle even a regular programming language like C can be used as intermediate step for generating code, but such an approach often comes with high compilation times. The HyPer system uses LLVM as backend, which is widely used (e.g., by the Clang C/C++ compiler), and supports most popular hardware platforms (e.g., x86, ARM, PowerPC). We found it to be mature and efficient.

But while a compiler backend allows for generating "raw" machine code, additional abstraction layers are very helpful to simplify mapping SQL queries into machine code. The most important ones are a typed value system, high-level control flow logic, and tuple storage. In the following, we will look at these layers individually.

3.1 LLVM API

The LLVM assembly language can be considered a machine language for an abstract machine – similar to the Java bytecode being the language of the Java Virtual Machine (JVM). However, it is more low-level than the Java bytecode as it closely matches commonly used CPUs. Nevertheless, it abstracts away many of the idiosyncratic features of real CPUs, and offers a clean and well-defined interface.

To generate instructions, LLVM offers a convenient C++ API. HyPer is written in C++ and can therefore directly use this API. The code to generate an integer addition, for example, is as follows:

```
llvm::Value* result=codegen->CreateAdd(a,b);
```

The variables `a`, `b`, and `result`, which have type `llvm::Value*`, are compile-time handles to *registers*. In contrast to registers in real machines, the LLVM abstract machine has an unbounded number of registers with symbolic names; these are mapped to real CPU registers by the platform-specific LLVM backends. This low-level interface is close to the hardware, but for most steps of the query compilation process we prefer a richer interface that exposes high-level functionality.

3.2 Typed Values

SQL defines a number of data types, which must be mapped to the low-level LLVM data types. Figure 3a shows this mapping for some common data types. A *null-able* `integer` value, for example, is represented using two

types, a 1-bit integer (`int1`), which indicates if the value is null, and a 32-bit integer (`int32`), which stores the actual value. Of course, data types that are declared as `not null` do not require the additional null indicator, but only store the value, as shown in the `decimal(18,2) not null` example. This is possible because null checks are "compiled away" when the schema allows this.

Since SQL data types are generally represented using multiple LLVM values, it may seem natural to combine them in a single structure (record) in the generated code. However, this would lead to lower performance, because of additional pointer traversals to access the values in the structure. To keep as many values as possible in registers, we therefore combine the values *only at compile time* in the `Value` class. The class is shown in Figure 3b and represents a typed SQL value, which consists of its SQL type (`type` field) and one or more LLVM registers (of type `llvm::Value*`). The `null` field is not used if the data type is `not null`, and the `length` field is only used for variable length data like strings. Using this compile-time abstraction, an SQL value can be treated as an entity in the compiler, while still allowing to generate code that is as fast as possible.

SQL operators and functions are member functions of the `Value` class. For operators like + or * different code must be generated depending on the SQL type (e.g., different instructions are used for `integer` and `real`). The `Value` class also handles SQL's null semantics, type coercion (e.g., multiplication of `integer` and `real`), and provides additional functionality like hashing which is used internally. As a consequence, the implementation of relational operators can use this high-level interface to transparently operate on values with SQL data types instead of generating data type specific code.

Users of database systems rightly expect the result of queries to be mathematically correct. It is therefore not sufficient to simply map integer addition to the underlying machine instruction. It is also necessary to check for overflow, which is usually provided by the CPU as a side effect of the operation (e.g., carry flag). LLVM offers access to these flags, which we check after each operation that can fail. Note that in pure C there is no way to access these flags, so checking for overflows would have been more expensive if we had used C as a backend.

3.3 Control Flow

For control flow, the LLVM assembly language only provides conditional and unconditional branches. Furthermore, LLVM requires that the code has Static Single Assignment (SSA) form, i.e., each register can only be assigned once. SSA is a common program representation of optimizing compilers. While SSA form simplifies the implementation of LLVM, it can make the query compiler code quite intricate. We therefore introduce a number of easy-to-use high-level control flow constructs.

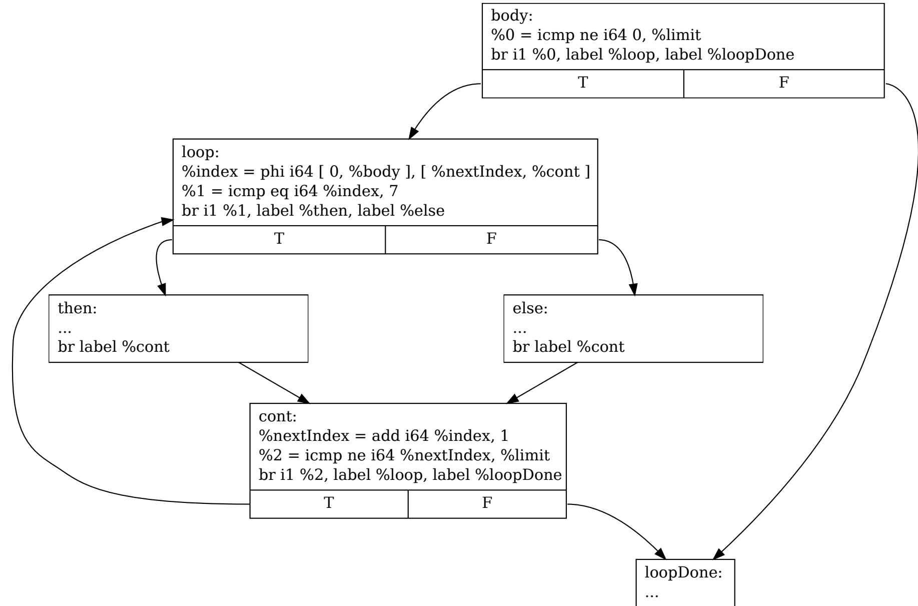
Figure 4a shows an example C program that contains two **if** statements and a **do-while** loop over the `index` variable. The LLVM representation of this program is shown in Figure 4b and consists of conditional branches between labeled code blocks, which are called *basic blocks*. As mentioned before, LLVM does not allow to change a register after it has been initialized, therefore it is not possible to simply assign a new value to the `index` register at each iteration step. Instead, a `phi` construct is used in the first line after the `loop` label. Using `phi`, a register can take different values depending on the preceding basic block. In our example, the `index` register gets the value 0 if the previous basic block was `body`, or the value `nextIndex` if the previous basic block was `cont`. It should be obvious that creating all these building blocks, branches, and phi nodes is quite tedious and requires a significant amount of code if done manually (for the example about 20 lines are needed).

The high-level `If` and `FastLoop` constructs allow to generate the control flow graph in Figure 4b much more easily:

```
1  llvm::Value* initialCond=codegen->CreateICmpNE(codegen.const64(0),limit);
2  FastLoop loopIndex(codegen,"",initialCond,{{codegen.const64(0),"index"}});
3  {
4    llvm::Value* index=loopIndex.getLoopVar(0);
5    {
6      If checkSeven(codegen,codegen->CreateICmpEQ(index,codegen.const64(7)));
```

```
uint64_t index=0;
if (index!=limit) do {
  if (index==7) {
    // do this ...
  } else {
    // do that ...
  }
  index++;
} while (index!=limit);
```

(a) Example C code



(b) LLVM control flow graph for (a)

Figure 4: Control flow example

```
7        // do this ...
8        checkSeven.elseBlock();
9        // do that ...
10     }
11     llvm::Value* nextIndex=codegen->CreateAdd(index,codegen.const64(1));
12     loopIndex.loopDone(codegen->CreateICmpNE(nextIndex,limit),{nextIndex});
13  }
```

The `If` class generates control flow code for an if-then-else construct. The code after the constructor (line 6) is only executed if the condition is true. The (optional) else block starts after `elseBlock` has been called. The construct is terminated automatically in the destructor, which is called automatically if `checkSeven` goes out of scope (line 10). The `FastLoop` class generates an if-do-while style control flow and supports multiple iteration variables, which can be accessed using `getLoopVar`. `FastLoop` is very flexible and can be used for many loop patterns – without the need to create basic blocks, branches, and phi nodes manually.

It is important to note that this example code looks similar to the C code in Figure 4a. This simplifies the implementation considerably, as it allows to directly think in terms of the code one wants to generates, as opposed to low-level constructs like basic blocks and phi nodes. Of course, `FastLoop` and `If` are composeable and can be nested arbitrarily. In our experience, programmers new to the HyPer code base can easily pick up these constructs by example, without necessarily understanding LLVM in detail.

3.4 Tuple Abstraction

As mentioned in the introduction, we strive to keep attributes in registers if possible. Therefore, in our generated runtime code there are no tuples; we directly operate on registers storing attributes. However, since relational operators often operate tuple-at-a time, it is often convenient to materialize and dematerialize a set of attributes – in effect treating them as a tuple.

Depending on the use case we offer two storage formats for materialization. The `CompactStorage` format saves space by storing data compactly, e.g., if an integer is equal to null (at runtime) only a single bit

is used. As a consequence, tuples with the same type may have different sizes depending on their runtime values. This can be undesirable when an attribute is updated. Therefore, we provide another storage format, `UpdateableStorage`, which always reserves the maximum amount of space for each attribute. E.g., for a null-able integer, both the 1-bit null indicator and the 32-bit value are always stored. Both formats automatically compute the correct layout for alignment purposes and fast access.

Having such an abstraction greatly simplifies the implementation of higher-level operators like hash-joins, as these can now read and store complex tuples using one line of code. Internally the tuple materialization code is surprisingly complex due to the one-to-many mapping of SQL data types to LLVM values, and the different sizes and alignment restrictions of different data types. Writing this every time for every materializing operator is therefore not an option. In general it is highly advisable to use as many abstractions as possible, as these, by being only compile time abstractions, cause no runtime costs.

3.5 Operators

With the help of these building blocks, HyPer implements all operators required by SQL-92 including the join, aggregation, sort, and the set operators. For pragmatic reasons and to reduce compilation times, some operators are partially implemented in C++. We take care to generate query specific code, however, if the following two conditions hold: The code is (1) performance critical and (2) query specific. For example, the hash join code is mostly generated as it fulfills both criteria. However, spilling to disk during a join is so slow that this code can be implemented in C++. But even some performance critical algorithms like sorting are implemented in C++, as they are not really query specific. For sorting we only generate the comparison function that is used within the C++ sorting algorithm. Such an interaction between LLVM and C++ code is very easy, because both operate on the same data structures (in the same process) and can call each other without performance penalty.

4 Evaluation

To compare our *data-centric compilation* scheme with other approaches, we additionally implemented *block-wise processing*, and the *iterator model* (both compiled and interpreted). All models operated on exactly the same data structures and storage layout (column-wise storage). We used the following simple query as a microbenchmark:

```
select count(*) from R where a>0 and b>0 and c>0
```

By varying the number of filter conditions, which always evaluate to true, we obtain the following execution times (in ms):

	0	1	2	3
data-centric compilation	0.001	5.199	7.037	18.753
iterator model - compiled	3.283	16.009	28.185	40.475
iterator model - interpreted	3.279	30.317	58.701	90.299
block processing - compiled	10.97	13.129	20.001	26.292

Clearly, the interpreted iterator model is not competitive with compilation due to a huge number of function calls and bad locality. The compiled iterator model and block-wise processing have much better performance, but are still slower than the data-centric code. In these numbers we also see that the LLVM backend itself is an optimizing compiler: If the query has no predicate at all, the LLVM compiler transforms the "scan and add 1 for each tuple" code into "add up data chunk sizes", which is much faster, of course.

Figure 5 shows the results of another experiment where we executed the first 7 TPC-H queries on scale factor 100 (the results for the remaining 15 queries are qualitatively similar). Since this paper is about compilation,

TPC-H #	compile	executed code	HyPer generated	time in generated	runtime	Vectorwise runtime
1	13ms	5.6KB	42%	98%	9.0s	33.4s
2	37ms	10.9KB	58%	86%	2.4s	2.7s
3	15ms	6.6KB	36%	89%	27.5s	25.6s
4	12ms	6.2KB	33%	93%	21.6s	22.4s
5	23ms	8.1KB	42%	86%	31.4s	29.7s
6	5ms	1.1KB	82%	96%	5.5s	7.1s
7	31ms	9.4KB	51%	88%	23.8s	28.2s
geo. mean (all 22)	19ms	6.8KB	47%	81%	16.2s	21.5s

Figure 5: TPC-H results

we used single-threaded execution to measure code quality instead of scalability. As far as code generation is concerned, intra-query parallelism is largely orthogonal, because all threads execute the same, synchronized code. For HyPer, we show the execution and compilation times, the size of the machine code, the fraction of the machine code that was generated at runtime using LLVM, and the fraction of time spent in the generated code. For comparison, we also measured the execution times of the official TPC-H leader Vectorwise, which uses block-wise processing. The execution time of HyPer for query 1, which consists of a fast aggregation of a single table, is 3.7x faster than Vectorwise. The majority of the queries spent most time processing joins, so the performance of both systems is similar, though HyPer is usually faster.

Although some of the TPC-H queries are quite complex, the compilation times remain quite low (cf. column "compile"). In fact they are below the human perception threshold, so that users do not perceive any delay for ad hoc queries. Compiling C code, in contrast, takes seconds [10] and would cause an unnecessary perceptible delay. Note that we generate very compact code; the total size of all code executed by each query (cf. column "executed code") is significantly smaller than the instruction cache of modern CPUs (e.g., 32KB for Intel) – in contrast to traditional systems, where instruction cache misses can be a problem [14]. On average about half of the executed code is generated using LLVM (cf. column "generated"), the rest is pre-compiled C++ code fragments, which are being called from the generated code. However, most of the performance critical code paths are generated. In particular, this includes all data type specific code. Therefore, on average more than 80% of the executed CPU cycles are in the generated code (cf. column "time in generated").

5 Related Work

For query evaluation, most traditional disk-based database systems use the iterator model, which was proposed by Lorie [8] and popularized by Graefe [3]. However, if most or all of the data is in main memory, the interpretation overhead of the iterator model often becomes significant. Therefore, a number of approaches have been proposed to improve performance. The MonetDB system [9] materializes all intermediate results, which eliminates interpretation overhead at the cost of loosing pipelining completely. MonetDB/X100 [1], which was commercialized as Vectorwise, passes chunks of data (vectors) between operators, which amortizes the operator switching overhead. A detailed study that compares vectorized execution with compilation can be found in [13].

Another approach is to compile queries before execution. One early approach compiled queries to Java bytecode [12], which can be executed using the Java virtual machine. The HIQUE system compiles queries to C using code templates [6]. In this approach the operator boundaries are clearly visible in the resulting code. Hekaton [2], Microsoft SQL Servers's main-memory OLTP engine, compiles stored procedure into native machine code using C as an intermediate language. The compiler interface mimics the iterator model and

collapses a query plan into a single function that consists of labels and branches between them. In our HyPer system we generate data-centric LLVM code using the produce/consume model [10]. In this paper we show that in a compilation-based system abstraction does not need to incur a performance penalty, as these abstractions can be compiled away; this has also been observed by Koch [5].

6 Conclusions

We have presented how to compile code for data-centric query execution. Using the produce/consume model and the LLVM compiler backend, our HyPer systems compiles SQL queries to very efficient machine code. Additionally, we introduced a number of compile-time abstractions that simplify our query compiler by abstracting away many of the low-level details involved in code generation. As a result, our query compiler is maintainable and generates very efficient code.

References

[1] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[2] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD Conference*, pages 1243–1254, 2013.

[3] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[4] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[5] C. Koch. Abstraction without regret in data management systems. In *CIDR*, 2013.

[6] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[7] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM International Symposium on Code Generation and Optimization (CGO)*, pages 75–88, 2004.

[8] R. A. Lorie. XRM - an extended (n-ary) relational memory. *IBM Research Report*, G320-2096, 1974.

[9] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3):231–246, 2000.

[10] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4:539–550, 2011.

[11] N. Ramsey and S. L. P. Jones. A single intermediate language that supports multiple implementations of exceptions. In *PLDI*, pages 285–298, 2000.

[12] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using JVM. In *ICDE*, page 23, 2006.

[13] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, pages 33–40, 2011.

[14] P. Tözün, B. Gold, and A. Ailamaki. OLTP in wonderland: where do cache misses come from in major OLTP components? In *DaMoN*, 2013.