Bulletin of the Technical Committee on

Data Engineering

March 2014 Vol. 37 No. 1

IEEE Computer Society

Letters

Special Issue on When Compilers Meet Database Systems

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

More IEEE Computer Society Activities

I have taken a new role as co-chair of the Conferences Advisory Committee within the IEEE Computer Society. This led me to attend the Society's meeting in Long Beach, CA in February, and exposed me to the on-going efforts of the Society to adapt to changing circumstances.

Organizational time, of necessity, moves slowly compared with human time scales. Changing the Computer Society is, in some sense, like changing widely used software. Bad things happen if change is not done carefully. So the only news I have to report is that the people in the Computer Society, volunteers and staff, understand that change is required and a number of options are being considered.

I think there are three primary facets to the current Computer Society situation. (1) Financial: the decline in publication revenue requires rethinking the Computer Society "business model". (2) New technology: Mobility, cloud, social networks, and online activities in general create opportunities for an expanded role. (3) Technical Reputation: Any changes must preserve the reputation of the Computer Society as a "brand" synonymous with technical quality. That limits what can be done but enhances the value of the things that are done.

So the Computer Society is actively seeking a way forward. If you have thoughts on this, I would be very interested in hearing them. Working together is amazingly effective in such an endeavor.

The Current Issue

How languages "play" with database systems has always been an important issue. We (the database technical community) optimize queries, and then execute the queries using a database management system. This has not changed. We build database systems using languages. This has not changed. But.... in a very real sense, almost everything has changed.

One change is how much we want substantially higher performance for our queries and updates. This quest for performance has led to major re-architecting of database systems. Elements designed decades ago have been re-designed to work well on modern multi-core processors, exploiting enormous main memories, and sometimes using flash disks (SSDs). As performance of the system elements has increased, the fraction of query execution time that is actually consumed by the query specification/program using the database system has increased dramatically. This has led the database community to increase their exploitation of compiler technology.

Compiler generation of highly efficient query code is one aspect of the picture. Looking at the division of work between database system and application suggests that the boundary has some interesting flexibility that compiler technology might help us exploit. These are just two of the several threads to the work reported here. Sudarshan has assembled a cross-section of the work in this area, from query code generation to database system implementation, etc. This area has had a growth spurt that makes it both a very old and a very young area- characteristic of an area in turmoil, and primed to change the nature of our systems. This is a subject that really demands careful study and this issue is a great place to start.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

When Compilers Meet Databases

Although the worlds of databases and compilers have a long history of interaction, in recent years that have come together in interesting new ways.

First, there is a resurgence of interest in compiling queries to machine code. This idea actually dates back to the original System R prototype, but was abandoned early on since it was found to be too cumbersome and slow. New technologies, such as LLVM, which allow efficient just-in-time generation of optimized machine code have greatly improved the speed of compilation. Further, CPU is often the bottleneck, not only for main-memory databases (where it is to be expected, and which are becoming increasingly important), but also for many queries on disk-based databases. Compilation of queries to machine code can lead to significant performance benefits in such scenarios. The first two papers in this issue (by Neumann and Leis from TU Munich, and Viglas et al. from Edinburgh/Microsoft Research) describe key techniques for compiling queries to machine code. The next two papers (by Freedman et al. from Microsoft and by Wanderman-Milne and Li from Cloudera) describe how the Hekaton main-memory database engine, and Cloudera's Impala system compile queries to machine code, and the very impressive real-world benefits obtained from such compilation.

Second, static analysis and dynamic profiling of programs has been used very successfully, in recent years, to find bugs and security vulnerabilities and to detect performance problems. The fifth paper in this issue, by Chaudhuri et al. from Microsoft, describes tools using these techniques to assist database developers.

Third, there has been a lot of interest of late in optimizing database applications by analysis/rewriting of application programs. Poor performance of database applications is a significant real-world problem even today. Such problems are often due to programs accessing databases in inefficient ways, for example invoking a large number of database queries, each returning a small result, where the round-trip delays for each query quickly add up across multiple queries. Database query optimization cannot solve this problem. Programmers can be asked to rewrite their code, but that can be cumbersome and error prone. Object-relational mapping systems such as Hibernate, as well as language-integrated querying systems such as LINQ exacerbate this problem since they insulate programmers from the plumbing details of database access, making it easier to write applications that access databases in an inefficient manner.

The key to improving performance in such systems is the automated rewriting of application programs to improve performance. Such rewrite systems must first analyze the database interactions of the program using static analysis, and when possible rewrite the program m to improve the database access patterns. Two of the papers in this issue, by Cheung et al. from MIT/Cornell, and by Ramachandra and Guravannavar from IIT Bombay, describe several different ways to rewrite application programs to optimize database access. Techniques described in these papers can also potentially be used to optimize accesses to Web services.

Finally, the issue is rounded off by an eloquently written paper by Christoph Koch from EPFL, who argues that the time has come for (i) building database systems using high-level code, without worrying about efficiency, and (ii) building optimizing compilers that can generate efficient implementation from such high-level code. A similar paradigm has been very successful at the level of queries, leading to the dominance of SQL, but Christoph's article makes a strong case for applying it even to the task of building database systems.

I enjoyed reading all these articles, and I'm sure so will you.

S. Sudarshan
Indian Institute of Technology, Bombay

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.

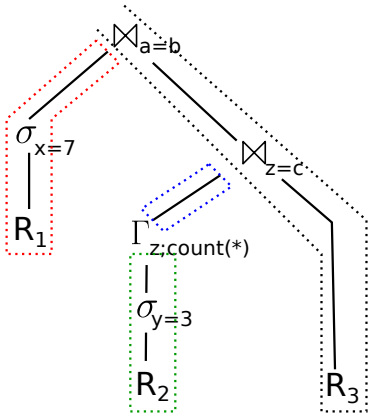
The iterator model worked well in the past, as query processing was dominated by I/O, so the overhead of the huge number of *next* calls was negligible. However, with increasing main memory sizes, this situation has changed. CPU and cache efficiency has become very important for good performance. This 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

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
select   *
from     R1,R3,
         (select   R2.z,
                   count(*)
          from     R2
          where    R2.y=3
          group by R2.z) R2
where    R1.x=7
and      R1.a=R3.b
and      R2.z=R3.c
```

(a) Example SQL Query



(b) Execution Plan

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z
for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$
for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z
for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$
for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}$ $[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}$ $[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$

(c) Compiled query (pseudo-code)

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.

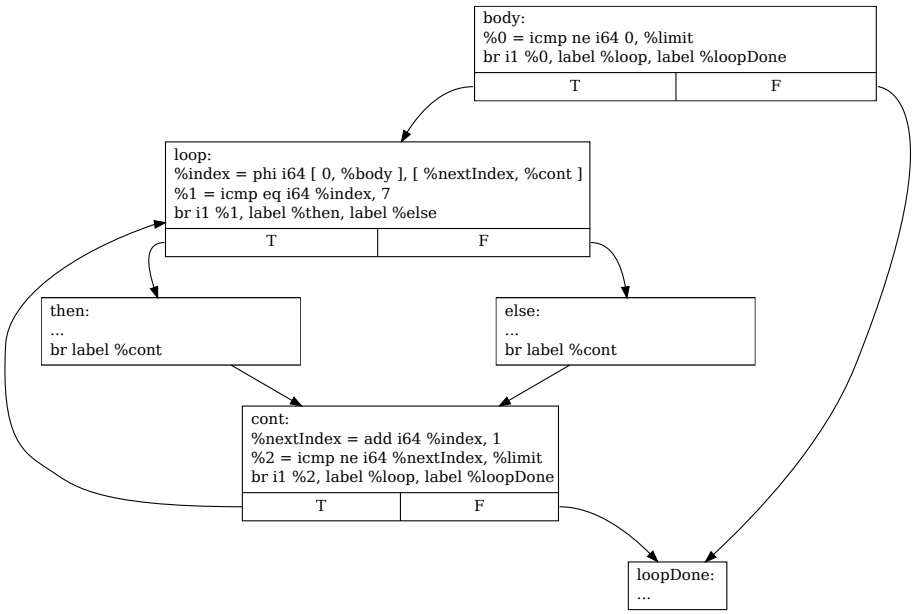
Figure 4a shows an example C program that contains two **if** statements and a **do-while** loop over the `index` variable. The LLVM representation of this program is shown in Figure 4b and consists of conditional branches between labeled code blocks, which are called *basic blocks*. As mentioned before, 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 #	HyPer compile	executed code	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.

Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes

Stratis D. Viglas
School of Informatics
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

Gavin Bierman
Microsoft Research
Cambridge, UK
gmb@microsoft.com

Fabian Nagel
School of Informatics
University of Edinburgh, UK
F.O.Nagel@sms.ed.ac.uk

Abstract

The falling price of main memory has led to the development and growth of in-memory databases. At the same time, language-integrated query has picked up significant traction and has emerged as a generic, safe method of combining programming languages with databases with considerable software engineering benefits. Our perspective on language-integrated query is that it combines the runtime of a programming language with that of a database system. This leads to the question of how to tightly integrate these two runtimes into one single framework. Our proposal is to apply code generation techniques that have recently been developed for general query processing. The idea is that instead of compiling quereies to query plans, which are then interpreted, the system generates customized native code that is then compiled and executed by the query engine. This is a form of just-in-time compilation. We argue in this paper that these techniques are well-suited to integrating the runtime of a programming language with that of a database system. We present the results of early work in this fresh research area. We showcase the opportunities of this approach and highlight interesting research problems that arise.

1 Introduction

Consider the architecture of a typical multi-tier application. The developer primarily decides on application logic: the data structures and algorithms to implement the core functionality. The data persistence layers of the application typically utilize a relational database system that has been optimized for secondary storage. It is accessed through its own query language (likely some variant of SQL) through bindings from the host programming language. The developer therefore has to deal with two different data models: (*a*) the application data model, which captures the data structures, algorithms, use-cases, and semantics of the application; and (*b*) the persistent data model, which captures the representation of data on secondary storage. An intermediate layer bridges the two data models and undertakes the cumbersome task of automating as much as possible of the translation between the two. The intermediate layer usually manifests as an API between the application programming language that accepts SQL strings as input; propagates them to the relational database for processing; retrieves the results; and pushes them back to the application for local processing. This clear separation of responsibility, functional as it may be, is potentially suboptimal in the context of the contemporary computing

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

environment. Just-in-time query compilation into native code aims to resolve this suboptimality. In this paper we examine the premises of this approach, showcase current work, and present opportunities for further work.

Our motivation stems from the observation that over the last few years we have been experiencing a slow but steady paradigm shift in the data management environment. The first factor contributing to this shift has been the continued reduction in price of main memory. This led to the working set of many data management applications fitting entirely into memory, and, hence, the development of in-memory databases (IMDBs). The development of IMDBs has addressed the efficiency problems of query processing over memory-resident data, as opposed to their on-disk counterparts. Work in this area has involved the development of new storage schemes (*e.g.*, column stores); new algorithms; and new database kernels that are optimized for main-memory I/O (*e.g.*, MonetDB). All these state-of-the-art solutions still make a key assumption: SQL is the entry point to the system.

The second factor contributing to the paradigm shift is the ever-tighter integration between SQL and the query specification mechanisms of host programming languages. As mentioned earlier, the mismatch between application data models and the relational model makes the mapping between the two cumbersome. The preprocessor-based approaches, *e.g.*, embedded SQL, substitute preprocessor macros with library calls and perform data type marshalling between the two runtimes; whereas library-based approaches are only slightly less intrusive by eschewing the preproccesor burden and having a tighter interface between SQL and platform types. But queries are still expressed in SQL and the library only undertakes type translation. *Language-integrated query* [10] is a relatively new class of techniques that rectifies this situation by enabling queries to be expressed using constructs of the host programming language. Developers have a uniform mechanism to pose queries over a disparate array of sources like web services, spreadsheets, and, of course, relational databases. The language-integrated query mechanism translates the query expressed in the host programming language to a format the source provider can process (*e.g.*, SQL if the provider is a relational database), transmits the query to the provider, retrieves the results, and converts them to host programming language types. This method feels more natural to the developer and is less error-prone. It does not solve, however, the fundamental problem of data being represented and potentially stored in two different runtimes: the host programming language and a remote source—a relational database in the majority of cases and the ones we focus on in this work.

We argue that in integrating querying between managed runtimes and in-memory database systems the best option is to blur the line between programming language and database system implementations, while, at the same time, borrowing ideas from compiler technology. Our stance is to internally fuse the two runtimes as much as possible so host-language information is propagated to the query engine; and database-friendly memory layouts and algorithms are used to implement the querying functionality. In what follows we will present the background for this line of research more extensively, including a brief overview of the current state-of-the-art (Section 2). We will then move on to exposing the synergy between the two runtimes (Section 3). We will present how the runtimes can be better integrated and showcase the open research directions in the area with an eye towards technologies to come (Section 4) before finally concluding our discussion (Section 5).

2 Background

Query processing has always involved striking a fine balance between the declarative and the procedural: managing the expressive power of a declarative language like SQL and mapping it to efficient and composable procedural abstractions to evaluate queries. Historically, relational database systems have compiled SQL into an intermediate representation: the query (or execution) plan. The query plan is composed of physical algebraic operators all communicating through a common interface: the iterator interface [13]. The query plan is then interpreted through continuous iterator calls to evaluate the query. This organization has many advantages for the database system: (*a*) it provides a high-level way to optimize queries through cost-based modeling; (*b*) it is extensible, as new operators can seamlessly be integrated so long as they implement the iterator interface; and (*c*) it is generic in its implementation of algorithms, which can be schema- and type-agnostic. This technique

has made SQL a well-optimized, but interpreted, domain-specific language for relational data management, and has formed the core of query engine design for more than thirty years.

Interpretation and macro-optimization. By catering for generality SQL interpretation primarily enables the use of macro-optimizations. That is, optimizations that can be performed at the query plan and operator levels, *e.g.*, plan enumeration strategies; cost modeling; algorithmic improvements; to name but a few of the better-known macro-optimizations. This view makes sense as SQL has been traditionally optimized for the boundary between on-disk and in-memory data, in other words I/O over secondary storage: choices at the macro level dramatically affect the performance of a query as they drastically change its performance profile.

The move to in-memory databases. In contemporary servers with large amounts of memory, it is conceivable for a large portion of the on-disk data—or even the entire database—to fit in main memory. In such cases, the difference in access latency between the processor's registers and main memory becomes the performance bottleneck [3]. To optimize execution one needs to carefully craft the code to minimize the processor stalls during query processing. Previous work [2, 28] has argued that this should be done from the ground up: changing the data layout to achieve enhanced cache locality, and then implementing query evaluation algorithms in terms of the new layout. The initial step in that direction was the further exploration of vertical decomposition [11] and the subsequent work on column stores [5] and query processing kernels optimized for main memory that have been built around them [28]. A stream of work in the area has introduced hybrid storage layouts [2]; vectorized execution for greater locality [6, 33]; and prefetching [7, 8, 9] to improve performance.

Code generation and execution. The advances in programming language design and implementation have been progressing independently of the work on relational database query processing. Some of the options in compiling source code in general and natively executing it are shown in Figure 1. Moving from left to right, the traditional option is to compile source code directly into native code to be executed by a host operating system (OS) on specific hardware; moving to different OS and hardware requires a version of the compiler for the target environment. Alternatively, the source code can be compiled into an intermediate representation termed bytecode that targets a virtual machine, as opposed to a physical one. Then an OS-specific implementation of the virtual machine, sometimes also referred to as a managed runtime, interprets the bytecode on the target OS and hardware. Part of the virtual machine is a managed heap: the space inside the virtual machine that is used for allocation of language-specific data structures. It is therefore the case that the virtual machine makes memory allocation calls to the OS, but then manages the language-specific data structures itself. This gives way to more advanced memory management techniques like garbage collection: the programmer need not explicitly account for memory allocation and deallocation as the virtual machine can track references to memory blocks internally and deallocate memory when it is no longer referenced. Moving to different OS and hardware now requires a different version of the virtual machine for the target environment, rather than a version of the compiler, as bytecode is OS- and hardware-independent. In a managed runtime, bytecode is interpreted as opposed to being natively executed. Just-in-time (JIT) compilation allows managed runtimes to convert blocks of bytecode to native code through a native compiler for the platform. JIT compilation

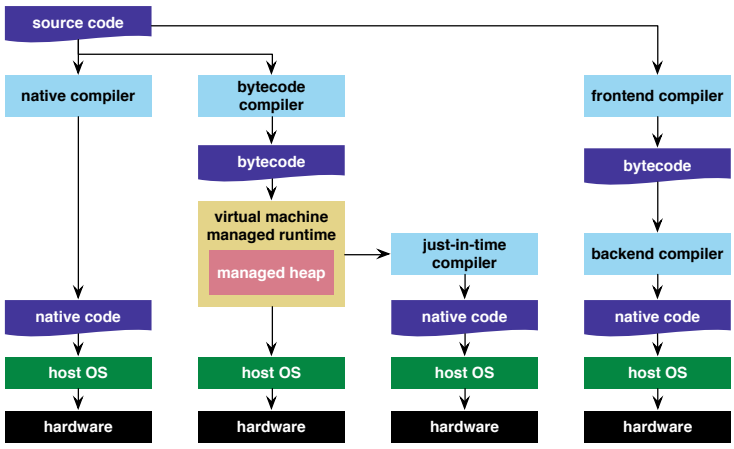


Figure 1: The options when compiling and executing source code: from source code to native code; from source code to bytecode to be executed in a managed runtime, either in an interpreted or in a just-in-time compiled way; from source code to bytecode for a virtual machine to be recompiled by a retargetable backend compiler.

is a powerful set of technologies that can lead to managed runtime performance that is comparable to a native implementation. There is a cost associated with JIT compilation as calling the compiler itself and dynamically binding the compiled code to the interpreted bytecode can be a more time-consuming operation than simply interpreting the bytecode. It is, however, a one-off cost and, for frequently executed blocks of code, that cost is amortized. Another option is to split compilation into two steps: a frontend compiler that translates source code into an intermediate bytecode representation; and a backend compiler that translates the intermediate representation into native code on demand and for each target OS and hardware. This means that the code can be executed on any environment for which a backend compiler exists, in contrast to relying on a virtual machine as was the case before. The first compilation step generates code for a register-based virtual machine that closely resembles contemporary processors and applies all code-specific and platform-independent optimizations possible (*e.g.*, register allocation and reuse). Whereas the backend compiler applies platform-specific optimizations that target the underlying OS and hardware. Effectively, such a stack turns JIT compilation in the principal way of code execution. The low-level virtual machine (LLVM) [27] is a typical example of such multi-stage compilation and optimization.

Relational databases as a managed runtime. There is an analogy to be drawn with query processing: SQL is effectively an interpreted language. The differences to standard programming language terminology is that we no longer interpret user programs but user queries. Additionally, the unit of translation is not a single statement or a code block, but potentially an entire query or an operator in a query tree. It is therefore no surprise that the revival of native code generation for SQL has started from applying these techniques in managed runtimes. Efforts in the area include the Daytona fourth generation language [14] that used on-the-fly code generation for high-level queries. This system, however, relied heavily on functionality that is traditionally handled by the database (*e.g.*, memory management, transaction management, I/O) to be provided by the underlying operating system. Similarly, the popular SQLite embedded database system, used an internal virtual machine as the execution mechanism for all database functionality and all operations are translated into virtual machine code to be executed [19]. Rao *et al.* [37] used the reflection API of Java to implement a native query processor targeting the Java virtual machine. Though again limited in its applicability, as it relied on Java support and was not an SQL processor but rather a Java-based query substrate for main memory.

3 Just-in-time compilation for SQL processing

An alternate route that has only recently begun to be explored is the application of micro-optimizations stemming from the use of standard compiler technology. That is, viewing SQL as a programming language and compiling it either into an intermediate representation to be optimized through standard compiler technology, or directly into native code. Such an approach does away with interpreting the query plan, blurs the boundaries between the operators of the iterator-centric solution, and collapses query optimization, compilation, and execution into a single unit. The result is a query engine that is free of any database-specific bloat that frequently accompanies generic solutions and, in a host of recent work, has exhibited exceptional performance.

In a strictly database context, past work has identified the generality of the common operator interface employed by the query engine, namely the *iterator model*, as the biggest problem with the dataflow of a database system [25]. The iterator model results in a poor utilization of the processor's resources. Its abstract implementation and its use of function calls inflates the total number of instructions and memory accesses required during query evaluation. One would prefer to make optimal use of the cache hierarchy and to reduce the load to the CPU and to the main memory. At the same time, one would not want to sacrifice the compositionality of the iterator model. To that end, Krikellas *et al.* [25] proposed *holistic query evaluation*. The idea is to inject a source code generation step in the traditional query evaluation process. The system looks at the entire query and optimises it holistically, by generating query-specific source code, compiling it for the host hardware, and then executing it. Using this framework, it is possible to develop a query engine that supports a substantial subset of SQL and is

highly efficient for main-memory query evaluation. The architecture of a holistic query engine is shown in Figure 2. The processing pipeline is altered through the injection of a code generation step after the query has been optimized. The output of the optimizer is a topologically sorted list O of operator descriptors o_i. Each o_i has as input either primary table(s), or the output of $o_j, j < i$. The descriptor contains the algorithm to be used in the implementation of each operator and additional information for initializing the code template of that algorithm. The entire list effectively describes a scheduled tree of physical operators since there is only one root operator. The optimized query plan is traversed and converted into C code in two steps per operator: (*a*) data staging, where the system performs all filtering operations and necessary preprocessing (*e.g.*, sorting or partitioning); and (*b*) holistic algorithm instantiation: the source code that implements the semantics of each operator. The holistic model includes algorithms for all major query operations like join processing, aggregation, and sorting. The code generator collapses operations where possible. For instance, in the presence of a group of operators that can be pipelined (*e.g.*, multiple pipelined joins, or aggregations over join outputs) the code generator nests them in a single code construct. The code is generated so there are no function calls to traverse the input; rather, traversal is through array access and pointer arithmetic. Once the code is generated, it is compiled by the C compiler, and the resulting binary is dynamically linked to the database engine. The latter loads the binary and calls a designated function to produce the query result. The nested, array-based access patterns that the generated code exhibits further aid both the compiler to generate efficient code at compile-time, and the hardware to lock on to the access pattern and issue the relevant prefetching instructions at run-time. The resulting code incurs a minimal number of function calls: all iterator calls are replaced with array traversals. Additionally, the cache miss profile of the generated code is close to that of query-specific hand-written code. These factors render holistic query processing as a high-performance, minimal-overhead solution for query processing. For example, in a prototype implementation, TPC-H Query 1 is reported to be over 150 times faster than established database technology.

The next fundamental piece of work on just-in-time compilation for SQL was by Neumann [31] and forms the basis of the query engine of the HyPer system [21], a hybrid main memory system targeting both OLTP and OLAP workloads. The proposal was to use compiler infrastructure for code generation; the resulting techniques were built on the LLVM framework. The main insight was that one should follow a data-centric as opposed to an operator-centric approach to query processing and thus depart from the traditional approach, which is based on evaluation plans with clear boundaries between operators. The high performance of these techniques further corroborates the wide applicability of code generation and just-in-time compilation of SQL queries [32]. These two systems prompted a number of further works. A comparison of just-in-time compilation to vectorized query processing based on vertical partitioning was undertaken by Sompolski *et al.* [40]. Their conclusion was that a hybrid solution can offer better performance. Zhang and Yang [42] applied polyhedral compilation primitives to optimize the I/O primitives in array analytics; while the techniques were different in terms of the compilation primitives, the results were similarly conducive to code generation as a viable query processing alternative. Kissinger *et al.* [22] applied the techniques for specialized processing on prefix trees, where the domain was much more controlled than full SQL evaluation. At a higher-level, Pirk *et al.* [35] ported the idea to general purpose GPUs. Part of a relational computation was specialized for and offloaded to the GPU in order to take advantage of the asymmetric memory channels between the two processors and improve memory I/O; the results were encouraging and showed performance improvements for key-foreign key joins. Murray *et al.* [29] applied a similar approach for LINQ [4]

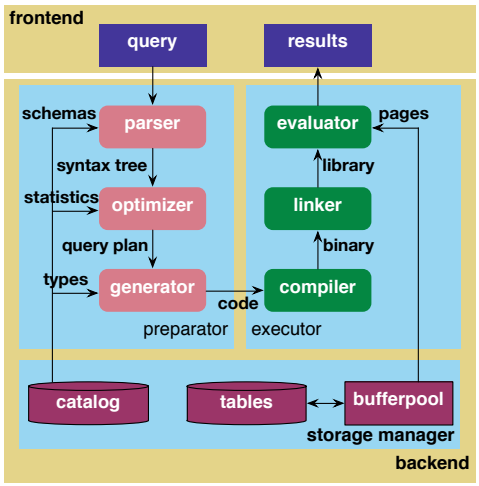


Figure 2: The architecture of HIQUE, the Holistic Integrated QUery Engine [25]

in the context of declarative processing, where the objective was to eliminate some of the bloat of a high-level runtime. As is true for all approaches based on code-generation the result was streamlined execution and a dramatic reduction in function invocations. Finally, the DBToaster project [1, 23] uses materialized views to store data and expresses SQL queries in an internal representation to incrementally evaluate the queries over the deltas of the updated primary data. The view maintenance mechanisms are converted to C++ or Scala code so they can be integrated into applications written in these languages.

4 The runtime is the database

Given the convergence of the two areas and the emergence of language-integrated query, it is reasonable to integrate the programming language and the database runtimes more tightly. The foremost reason is so data is not replicated in different data models across the two runtimes—especially in the context of IMDBs. Moreover, a tighter integration allows for improved type safety and less error-prone translations between data models. Likewise, an integrated runtime is more natural to the developer as it does not require them to think for two different runtime and data representations. Finally, it enables both language- *and* database-specific optimizations, resulting in a high-performance solution that is greater than the sum of its parts. Code generation and just-in-time compilation are the key technologies that can facilitate the transition to turn the managed runtime into a full-blown database solution. We will now present a list of research opportunities that arise from this tighter integration between programming language and database runtimes, all stemming from the introduction of (potentially just-in-time) code generation for the data management and query processing substrates.

Memory allocation and management. We first deal with the mismatch between the memory model of a programming language in a managed runtime and that of an IMDB. Memory allocation and deallocation in a managed runtime is triggered by application code, but is solely handled by the runtime itself through the managed heap (see also Figure 1). The developer declares types and requests instances of those types for the application. The runtime allocates memory and keeps track of references to that memory. Once instances are no longer referenced they can be deallocated from the managed heap and their memory reclaimed. On the other hand, database data is organized in records that reside in tables. This requires special treatment as the storage requirements and access patterns are not the ones typically found in general data structures. One approach is to extend the collection framework[1] found in most managed runtimes with table-specific types that will be managed in a more database-friendly way. This can be achieved by either creating table-aware collections (*e.g.*, a `Table` collection type) or table-aware element types (we call these tabular types), or both. This would allow the runtime to be aware of such types and treat them with database query processing in mind; at the same time, it would allow their seamless use in the rest of the managed runtime using the semantics of the host programming language. Code generation can automate the process either at compile-time by generating different representations for database collections, or at run-time by interjecting and customizing the representation to make it database-like.

Memory layout. Following on from the tabular type declaration, the next step is to represent such types in main memory. A table is effectively a collection of records. A typical way to represent such collections in managed runtimes would be through a collection type like an `ArrayList`. The in-memory representation of an `ArrayList` in a managed runtime, however, implies that, for any non-primitive types such as records, the array elements are references to objects allocated on the managed heap. This contrasts with the more familiar array representation in languages like C/C++ where data is laid out continuously in memory. The difference is shown in Figure 3. Doing away with references and data fragmentation reduces the number of cache misses, *e.g.*, in the case of an array traversal. At the same time, it allows the hardware to detect the access pattern and aggressively prefetch data. Note that some runtimes have support for more complicated value-types (*e.g.*, the `struct` type in C♯) which can then be grouped into arrays. However, there are restrictions on how they are processed by the runtime. When implementing dynamic arrays, the method of progressively doubling the array on expansion which is commonly

[1] These are libraries of abstract data types like linked-lists, arrays, and search trees used to store homogeneous collections of objects.

found in programming languages (*e.g.*, Java and C♯ `ArrayList`s, or even C++ `vector`s) is not the best option as it results in a high data copying overhead. Instead, a blocked memory layout is more beneficial and such techniques can be incorporated when integrating the programming language and database runtimes [5, 28, 41]. Inlining techniques are not only relevant to arrays as a principal data organization. They are useful in other auxiliary data structures like indexes where saving multiple pointer/reference traversals through careful inlining will result in substantial accumulated savings. Either the compiler can statically determine optimization opportunities, or the runtime can dynamically customize the layout through just-in-time compilation.

Data access semantics and decomposition model. Managed runtimes can further benefit from a workload-driven representation of data in memory. For instance, one might consider hints by the developer to designate key constraints over a tabular collection. Alternatively, either through developer hints or through statically analyzing the code, the compiler can detect cases where vertical decomposition or a different grouping of fields in a tabular collection is more beneficial and generate the appropriate code. Consider the following suggestive code fragment:

```
public tabular order {
  public key int number;
  public access date orderdate;
  group { public int quantity; public string item; }}
```

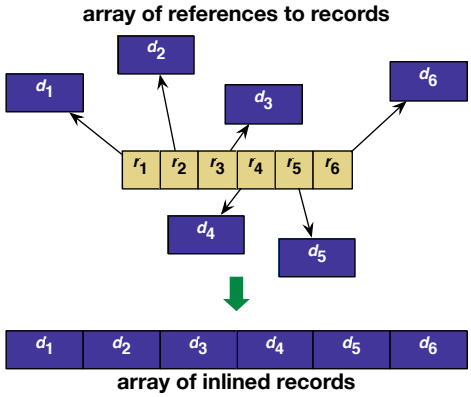


Figure 3: Standard *vs.* inlined representation of tabular data; the second option is closer to an IMDB memory layout.

where, in defining the new tabular type (marked with a keyword `tabular`), we also specify a key constraint on the `number` field; give an access method hint on the `date` field and exclude these two fields from the main group to indicate to the runtime that they are frequently accessed individually (*e.g.*, in selections). The compiler and the runtime can use this information to build auxiliary structures and/or alter the main memory layout for the elements of this collection. For instance, indexes may be automatically built on the `number` and `date` attributes: the first one to enforce the key constraint, the second one to improve performance. Likewise, it may well be the case that the system chooses to employ a partially decomposed storage model *à la* Hyrise [15] which mixes row- and column-based storage to maximize performance (*e.g.*, storing `number` and `date` in a columnar fashion). Note that work in coupling JIT compilation with partial decomposition has been undertaken in the context of database query processing and has produced encouraging results [34].

Code caching. One underlying assumption is that the compilation cost can be amortized, as, for short-running queries, compilation time may be in the same order as query processing time [25, 31]. If the workload is static this is not a problem as compilation is a one-off cost. But if the workload is dynamic, then compilation becomes a bottleneck. One future work direction is to provide adaptive solutions to automatically identify the queries that are good candidates for compilation, in truly JIT-like form. Additionally, in a dynamic system, compiled queries take up space in the memory. It is then conceivable to couple the JIT compilation decisions with admission control policies where the system manages a fixed memory budget for compiled queries and decides when queries are admitted and evicted from the compiled query pool. Work on intermediate and final result caching [12, 20, 24, 30, 39] and batch-based multi-query optimization [38] may be of benefit here.

Manycore processing. Code generation for query processing in managed runtimes may be of benefit in, potentially heterogeneous, manycore systems. Krikellas *et al.* [26] took a first step towards this with multithreaded processing on multicore CPUs, though the JIT compiler was mainly a tool and not the primary object of study. Code generation can be helpful for parallel processing as it allows fine parallelization orchestration without relying on generic parallelization solutions through coarse-grained techniques. At the same time, it fits well with work on JIT-compiled approaches to heterogeneous manycore runtimes like OpenCL [17]. It seems natural to explore the potential for synergy between these approaches for SQL query processing.

Non-volatile memory. Non-volatile, or persistent, memory enables the persistence of data stored in the managed heap. Non-volatile memory (NVM) is a new class of memory technology with the potential to deliver on the promise of a universal storage device. That is, a storage device with capacity comparable to that of hard disk drives; and access latency comparable to that of random access memory (DRAM). Such a medium blurs even further the line between application programming and data management and will most likely solidify the need for extending managed runtimes with data management and query processing capabilities without any need for offloading processing to a relational database. Performance-wise, non-volatile memory sits between flash memory and DRAM: its read latency is only 2-4 times slower than DRAM compared to the 32 times slower-than-DRAM latency of flash [36]. However, NVM is byte-addressable, which is in stark contrast to block-addressable flash memory. At the same time, NVM exhibits the write performance problems of flash memory: writes are more than one order of magnitude slower than DRAM, and thus more expensive than reads. Persistent memory cells also have limited endurance, which dictates wear-leveling data moves across the device to increase its lifetime, thereby further amplifying write degradation. Recent work has argued that in the presence of persistent memory one should cater for the read/write asymmetry by specifying and dynamically altering the write-intensity of the processing algorithms depending on the workload [41]. It is therefore sensible to inject these dynamic decisions to the runtime through appropriate JIT compilation techniques.

Transactional processing. Another potential direction is transaction processing and concurrency control. That is not to say that radically different concurrency control mechanisms are needed for JIT-compiled queries; quite the contrary, one of the reasons that the approach works so well is that it does not affect orthogonal system aspects. One can, however, compile the concurrency control primitives themselves to a lower-level for more efficient code without compromising integrity. One possibility is to create workload-specific locking protocols and then automatically generate code for them. An alternative is to compile concurrency into hardware transactional memory primitives [16, 18] and thus further customize the code for the host hardware.

5 Conclusions and outlook

Code generation for query processing is a new and potentially game-changing approach to a core database use-case: integrating database systems with programming languages. Prior work in the context of code generation for SQL processing has exhibited significant performance improvements over traditional techniques and, as such, the approach is very promising. It is therefore sensible to aim for a tighter integration between the runtime of a programming language and that of an in-memory database system. We have argued that the more natural way to achieve this is through code generation and native support for the forms of just-in-time compilation found in most contemporary managed runtimes. We have showcased the state-of-the-art work in this area and showed how it enables highly efficient query processing. As is the case with any new research area, especially one that drops fundamental assumptions, there are quite a few open research questions. We have provided a list of such open questions with an eye towards improving the support for processing of a declarative query language like SQL in the context of a general purpose programming language with language-integrated query functionality.

Acknowledgments. This work was supported by a Microsoft Research PhD Scholarship and the Intel University Research Office through the Software for Persistent Memories program.

References

[1] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10), 2012.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, 1999.

[4] G. M. Bierman, E. Meijer, and M. Torgersen. Lost In Translation: Formalizing Proposed Extensions to C#. In *OOPSLA*, 2007.

[5] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, 2002.

[6] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[7] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB*, 2005.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), 2007.

[9] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD*, 2001.

[10] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.

[11] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[12] S. Finkelstein. Common expression analysis in database applications. In *SIGMOD*, 1982.

[13] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2), 1993.

[14] R. Greer. Daytona And The Fourth-Generation Language Cymbal. In *SIGMOD*, 1999.

[15] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A main memory hybrid storage engine. *PVLDB*, 4(2), 2010.

[16] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[17] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9), 2013.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[19] D. R. Hipp, D. Kennedy, and J. Mistachkin. SQLite database. `http://www.sqlite.org`. Online; accessed February 2014.

[20] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. In *SIGMOD*, 2009.

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

[22] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. QPPT: Query processing on prefix trees. In *CIDR*, 2013.

[23] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[24] Y. Kotidis and N. Roussopoulos. Dynamat: a dynamic view management system for data warehouses. In *SIGMOD*, 1999.

[25] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[26] K. Krikellas, S. D. Viglas, and M. Cintra. Modeling multithreaded query execution on chip multiprocessors. In *ADMS*, 2010.

[27] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[28] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9, 2000.

[29] D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic Optimization of Declarative Queries. In *PLDI*, 2011.

[30] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *ICDE*, 2013.

[31] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9), 2011.

[32] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Engineering Bulletin*, 37(1), 2014.

[33] S. Padmanabhan, T. Malkemus, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

[34] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. L. Kersten. CPU and cache efficient management of memory-resident databases. In *ICDE*, 2013.

[35] H. Pirk, S. Manegold, and M. Kersten. Accelerating Foreign-Key Joins using Asymmetric Memory Channels. In *ADMS*, 2011.

[36] M. K. Qureshi, S. Gurumurthi, and B. Rajendran. *Phase Change Memory: from devices to systems*. Morgan & Claypool Publiishers, 2012.

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

[38] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[39] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Inf. Syst.*, 13, 1988.

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

[41] S. D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[42] Y. Zhang and J. Yang. Optimizing I/O for Big Array Analytics. *PVLDB*, 5(8), 2012.

Compilation in the Microsoft SQL Server Hekaton Engine

Craig Freedman
craigfr@microsoft.com

Erik Ismert
eriki@microsoft.com

Per-Ake Larson
palarson@microsoft.com

Abstract

Hekaton is a new database engine optimized for memory resident data and OLTP workloads that is fully integrated into Microsoft SQL Server. A key innovation that enables high performance in Hekaton is compilation of SQL stored procedures into machine code.

1 Introduction

SQL Server and other major database management systems were designed assuming that main memory is expensive and data resides on disk. This assumption is no longer valid; over the last 30 years memory prices have dropped by a factor of 10 every 5 years. Today, one can buy a server with 32 cores and 1TB of memory for about $50K and both core counts and memory sizes are still increasing. The majority of OLTP databases fit entirely in 1TB and even the largest OLTP databases can keep the active working set in memory.

Recognizing this trend SQL Server several years ago began building a database engine optimized for large main memories and many-core CPUs. The new engine, code named Hekaton [2][3], is targeted for OLTP workloads.

Several main memory database systems already exist, both commercial systems [4][5][6][7][8] and research prototypes [9][10][11][12]. However, Hekaton has a number of features that sets it apart from the competition.

Most importantly, the Hekaton engine is integrated into SQL Server; it is not a separate DBMS. To take advantage of Hekaton, all a user has to do is declare one or more tables in a database memory optimized. This approach offers customers major benefits compared with a separate main-memory DBMS. First, customers avoid the hassle and expense of another DBMS. Second, only the most performance-critical tables need to be in main memory; other tables can be left unchanged. Third (and the focus of this article), stored procedures accessing only Hekaton tables can be compiled into native machine code for further performance gains. Fourth, conversion can be done gradually, one table and one stored procedure at a time.

Memory optimized tables are managed by Hekaton and stored entirely in main memory. Hekaton tables can be queried and updated using T-SQL in the same way as regular SQL Server tables. A query can reference both Hekaton tables and regular tables and a single transaction can update both types of tables. Furthermore, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code. This is by far the fastest way to query and modify data in Hekaton tables and is essential to achieving our end-to-end performance goals for Hekaton.

The rest of the article is organized as follows. Section 2 outlines the high-level considerations and principles behind the design of Hekaton. Section 3 describes how stored procedures and table definitions are compiled into native code. Section 4 provides some experimental results.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Terminology. We will use the terms Hekaton table and Hekaton index to refer to tables and indexes stored in main memory and managed by Hekaton. Tables and indexes managed by the traditional SQL Server engine will be called regular tables and regular indexes. Stored procedures that have been compiled to native machine code will simply be called compiled stored procedures and traditional non-compiled stored procedures will be called interpreted stored procedures.

2 Design Considerations

Our goal at the outset of the Hekaton project was to achieve a 10-100X throughput improvement for OLTP workloads. An analysis done early on in the project drove home the fact that a 10-100X throughput improvement cannot be achieved by optimizing existing SQL Server mechanisms. Throughput can be increased in three ways: improving scalability, improving CPI (cycles per instruction), and reducing the number of instructions executed per request. The analysis showed that, even under highly optimistic assumptions, improving scalability and CPI can produce only a 3-4X improvement.

The only real hope is to reduce the number of instructions executed but the reduction needs to be dramatic. To go 10X faster, the engine must execute 90% fewer instructions and yet still get the work done. To go 100X faster, it must execute 99% fewer instructions. This level of improvement is not feasible by optimizing existing storage and execution mechanisms. Reaching the 10-100X goal requires a much more efficient way to store and process data.

So to achieve 10-100X higher throughput, the engine must execute drastically fewer instructions per transaction, achieve a low CPI, and have no bottlenecks that limit scalability. This led us to three architectural principles that guided the design.

2.1 Optimize indexes for main memory

Current mainstream database systems use disk-oriented storage structures where records are stored on disk pages that are brought into memory as needed. This requires a complex buffer pool where a page must be protected by latching before it can be accessed. A simple key lookup in a B-tree index may require thousands of instructions even when all pages are in memory.

Hekaton indexes are designed and optimized for memory-resident data. Durability is ensured by logging and checkpointing records to external storage; index operations are not logged. During recovery Hekaton tables and their indexes are rebuilt entirely from the latest checkpoint and logs.

2.2 Eliminate latches and locks

With the growing prevalence of machines with 100's of CPU cores, achieving good scaling is critical for high throughput. Scalability suffers when the systems has shared memory locations that are updated at high rate such as latches and spinlocks and highly contended resources such as the lock manager, the tail of the transaction log, or the last page of a B-tree index [13][14].

All of Hekaton's internal data structures, for example, memory allocators, hash and range indexes, and the transaction map, are entirely latch-free (lock-free). There are no latches or spinlocks on any performance-critical paths in the system. Hekaton uses a new optimistic multiversion concurrency control algorithm to provide transaction isolation semantics; there are no locks and no lock table [15]. The combination of optimistic concurrency control, multiversioning and latch-free data structures results in a system where threads execute without stalling or waiting.

2.3 Compile requests to native code

SQL Server uses interpreter based execution mechanisms in the same ways as most traditional DBMSs. This provides great flexibility but at a high cost: even a simple transaction performing a few lookups may require several hundred thousand instructions.

Hekaton maximizes run time performance by converting statements and stored procedures written in T-SQL into customized, highly efficient native machine code. The generated code contains exactly what is needed to execute the request, nothing more. As many decisions as possible are made at compile time to reduce runtime overhead. For example, all data types are known at compile time allowing the generation of efficient code.

The importance of compiling stored procedures to native code is even greater when the instruction path length improvements and performance improvements enabled by the first two architectural principles are taken into account. Once other components of the system are dramatically sped up, those components that remain unimproved increasingly dominate the overall performance of the system.

3 Native Compilation in Hekaton

As noted earlier, a key architectural principle of Hekaton is to perform as much computation at compile time as possible. Hekaton maximizes run time performance by converting SQL statements and stored procedures into highly customized native code. Database systems traditionally use interpreter based execution mechanisms that perform many run time checks during the execution of even simple statements. For example, the Volcano iterator model [1] uses a relatively small number of query operators to execute any query. Each iterator by definition must be able to handle a wide variety of scenarios and cannot be customized to any one case.

Our primary goal is to support efficient execution of compile-once-and-execute-many-times workloads as opposed to optimizing the execution of ad hoc queries. We also aim for a high level of language compatibility to ease the migration of existing SQL Server applications to Hekaton tables and compiled stored procedures. Consequently, we chose to leverage and reuse technology wherever suitable. We reuse much of the SQL Server T-SQL compilation stack including the metadata, parser, name resolution, type derivation, and query optimizer. This tight integration helps achieve syntactic and semantic equivalence with the existing SQL Server T-SQL language. The output of the Hekaton compiler is C code and we leverage Microsoft's Visual C/C++ compiler to convert the C code into machine code.

While it was not a goal to optimize ad hoc queries, we do want to preserve the ad hoc feel of the SQL language. Thus, a table or stored procedure is available for use immediately after it has been created. To create a Hekaton table or a compiled stored procedure, the user merely needs to add some additional syntax to the CREATE TABLE or CREATE PROCEDURE statement. Code generation is completely transparent to the user.

Figure 1 illustrates the overall architecture of the Hekaton compiler. There are two main points where we invoke the compiler: during creation of a Hekaton table and during creation of a compiled stored procedure.

As noted above, we begin by reusing the existing SQL Server compilation stack. We convert the output of this process into a data structure called the mixed abstract tree or MAT. This data structure is a rich abstract syntax tree capable of representing metadata, imperative logic, expressions, and query plans. We then transform the MAT into a second data structure called the pure imperative tree or PIT. The PIT is a much "simpler" data structure that can be easily converted to C code (or theoretically directly into the intermediate representation for a compiler backend such as Phoenix [17] or LLVM [16]). We discuss the details of the MAT to PIT transformation further in Section 3.2. Once we have C code, we invoke the Visual C/C++ compiler and linker to produce a DLL. At this point it is just a matter of using the OS loader to bring the newly generated code into the SQL Server address space where it can be executed.

3.1 Schema Compilation

It may not be obvious why table creation requires code generation. In fact, there are two reasons why table creation requires code generation.

The first reason is that the Hekaton storage engine treats records as opaque objects. It has no knowledge of the internal content or format of records and cannot directly access or process the data in records. The Hekaton compiler provides the engine with customized callback functions for each table. These functions perform tasks such as computing a hash function on a key or record, comparing two records, and serializing a record into a log buffer. Since these functions are compiled into native code, index operations such as inserts and searches are extremely efficient.

The second reason is that SQL Server's interpreted query execution engine can be leveraged to access Hekaton tables in queries that are not part of a compiled stored procedure. We refer to this method of accessing Hekaton tables as interop. While interop leverages the interpreted query execution engine code and operators, it does require some mechanism to crack Hekaton records and extract column values. When a new table is created, the Hekaton compiler determines the record layout and saves information about this record layout for use by the interop code. We discuss interop further in Section 3.4.

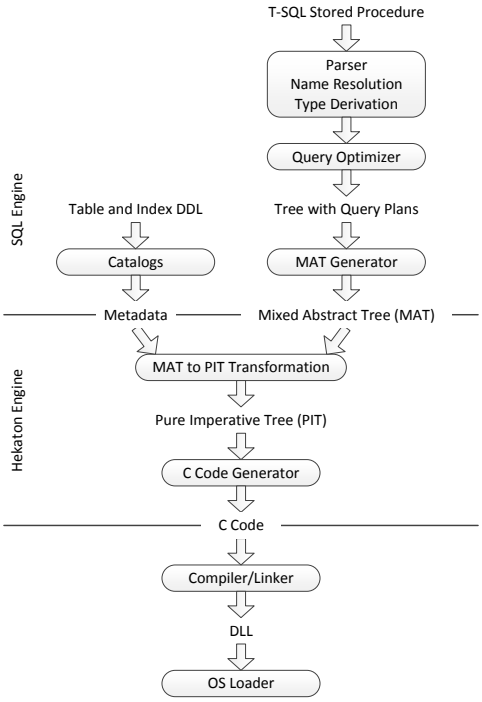


Figure 1: Architecture of the Hekaton compiler.

3.2 Stored Procedure Compilation

There are numerous challenging problems that we had to address to translate T-SQL stored procedures into C code. Perhaps the most obvious challenge is the transformation of query plans into C code and we will discuss our approach to this problem momentarily. There are, however, many other noteworthy complications. For example, the T-SQL and C type systems and expression semantics are very different. T-SQL includes many data types such as date/time types and fixed precision numeric types that have no corresponding C data types. In addition, T-SQL supports NULLs while C does not. Finally, T-SQL raises errors for arithmetic expression evaluation failures such as overflow and division by zero while C either silently returns a wrong result or throws an OS exception that must be translated into an appropriate T-SQL error.

These complexities were a major factor in our decision to introduce the intermediate step of converting the MAT into the PIT rather than directly generating C code. The PIT is a data structure that can be easily manipulated, transformed, and even generated out of order in memory. It is much more challenging to work directly with C code in text form.

The transformation of query plans into C code warrants further discussion. To aid in this discussion, consider the simple T-SQL example in Figure 2. This procedure retrieves a customer name, address, and phone number given a customer id. The procedure declaration includes some additional syntax; we will explain below why this syntax is required.

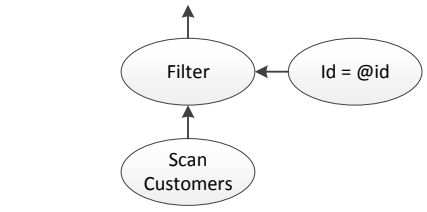
As with many query execution engines, we begin with a query plan which is constructed out of operators such as scans, joins, and aggregations. Figure 3(a) illustrates one possible plan for executing our sample query. For this example, we are naively assuming that the DBA has not created an index on Customer.Id and that the

predicate is instead evaluated via a filter operator. In practice, we ordinarily would push the predicate down to the storage engine via a callback function. However, we use the filter operator to illustrate a more interesting outcome.

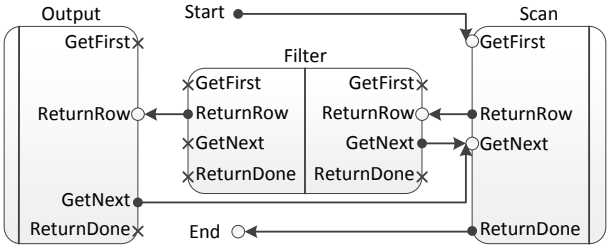
```
CREATE PROCEDURE SP_Example @id INT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT,
                      LANGUAGE = 'English')
   SELECT Name, Address, Phone FROM dbo.Customers WHERE Id = @id
END
```

Figure 2: Sample T-SQL procedure.

Each operator implements a common interface so that they can be composed into arbitrarily complex plans. In our case, this interface consists of GetFirst, GetNext, ReturnRow, and ReturnDone. However, unlike most query execution engines, we do not implement these interfaces using functions. Instead, we collapse an entire query plan into a single function using labels and gotos to implement and connect these interfaces. Figure 3(b) illustrates graphically how the operators for our example are interconnected. Each hollow circle represents a label while each arrow represents a goto statement. In many cases, we can directly link the code for the various operators bypassing intermediate operators entirely. The X's mark labels and gotos that have been optimized out in just such a fashion. In conventional implementations, these same scenarios would result in wasted instructions where one operator merely calls another without performing any useful work.



(a) Query plan



(b) Operator interconnects

Figure 3: Query plan and operator interconnects for sample T-SQL procedure.

Execution of the code represented by Figure 3(b) begins by transferring control directly to the GetFirst entry point of the scan operator. Note already the difference as compared to traditional query processors which typically begin execution at the root of the plan and invoke repeated function calls merely to reach the leaf of the tree even when the intermediate operators have no work to do. Presuming the Customers table is not empty, the scan operator retrieves the first row and transfers control to the filter operator ReturnRow entry point. The filter operator evaluates the predicate and either transfers control back to the scan operator GetNext entry point if the current row does not qualify or to the output operator ReturnRow entry point if the row qualifies. The output operator adds the row to the output result set to be returned to the client and then transfers control back to the scan operator GetNext entry point again bypassing the filter operator. When the scan operator reaches the end of the table, execution terminates immediately. Again control bypasses any intermediate operators.

This design is extremely flexible and can support any query operator including blocking (e.g., sort and group by aggregation) and non-blocking (e.g., nested loops join) operators. Our control flow mechanism is also flexible enough to handle operators such as merge join that alternate between multiple input streams. By keeping all of the generated code in a single function, we avoid costly argument passing between functions and expensive function calls. Although the resulting code is often challenging to read due in part to the large number of goto statements, it is important to keep in mind that our intent is not to produce code for human consumption. We rely on the compiler to generate efficient code. We have confirmed that the compiler indeed does so through inspection of the resulting assembly code.

Figure 4 gives a sample of the code produced for the sample procedure from Figure 2. We show only the code generated for the seek and filter operators with some minor editing for the sake of both brevity and clarity.

While the generated code ordinarily has no comments to protect against security risks and potential ”C injection attacks” (see Section 3.3), we do have the ability to add comments in internal builds for development and supportability purposes. The code in Figure 4 includes these comments for the sake of clarity. A careful analysis of this code sample will show that all of the gotos and labels are just as described above.

We compared this design to alternatives involving multiple functions and found that the single function design resulted in the fewest number of instructions executed as well as the smallest overall binary. This result was true even with function inlining. In fact, the use of gotos allows for code sharing within a single function. For example, an outer join needs to return two different types of rows: joined rows and NULL extended rows. Using functions and inlining with multiple outer joins, there is a risk of an exponential growth in code size [18]. Using gotos, the code always grows linearly with the number of operators.

```
    /*Seek*/
l_17:; /*seek.GetFirst*/
    hr = (HkCursorHashGetFirst(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->Transaction),
        0, 0, 1,
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
l_20:; /*seek.1*/
    if ((hr == 0))
    {
        goto l_14 /*filter.child.ReturnRow*/ ;
    }
    else
    {
        goto l_12 /*query.ReturnDone*/ ;
    }
l_21:; /*seek.GetNext*/
    hr = (HkCursorHashGetNext(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->ErrorObject),
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
    goto l_20 /*seek.1*/ ;
    /*Filter*/
l_14:; /*filter.child.ReturnRow*/
    result_22 = ((rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ->hkc_1 /*[Id]*/ ) ==
        ((long)((valueArray[1 /*@id*/ ]).SignedIntData)));
    if (result_22)
    {
        goto l_13 /*output.child.ReturnRow*/ ;
    }
    else
    {
        goto l_21 /*seek.GetNext*/ ;
    }
```

Figure 4: Sample of generated code.

There are cases where it does not make sense to generate custom code. For example, the sort operator is best implemented using a generic sort implementation with a callback function to compare records. Some functions (e.g., non-trivial math functions) are either sufficiently complex or expensive that it makes sense to include them in a library and call them from the generated code.

3.3 Restrictions

With minor exceptions, compiled stored procedures look and feel just like any other T-SQL stored procedures. We support most of the T-SQL imperative surface area including parameter and variable declaration and assignment as well as control flow and error handling (IF, WHILE, RETURN, TRY/CATCH, and THROW). The query surface area is a bit more limited but we are expanding it rapidly. We support SELECT, INSERT, UPDATE, and DELETE. Queries currently can include filters, inner joins, sort and top sort, and basic scalar and group by aggregation.

In an effort to minimize the number of run time checks and operations that must be performed each time a compiled stored procedure is executed, we do impose some requirements.

First, unlike a conventional stored procedure which upon execution can inherit run time options from the user’s environment (e.g., to control the behavior of NULLs, errors, etc.), compiled stored procedures support a very limited set of options and those few options that can be controlled must be set at compile time only. This policy both reduces the complexity of the code generator and improves the performance of the generated code by eliminating unnecessary run time checks.

Second, compiled stored procedures must execute in a security or user context that is predefined when the procedure is created rather than in the context of the user who executes the procedure. This requirement allows us to run all permission checks once at procedure creation time instead of once per execution.

Third, compiled stored procedures must be schema bound. This restriction means that once a procedure is created, any tables referenced by that procedure cannot be dropped without first dropping the procedure. This requirement avoids the need to acquire costly schema stability locks before executing the procedure.

Fourth, compiled stored procedures must execute in the context of a single transaction. This requirement is enforced through the use of the BEGIN ATOMIC statement (and the prohibition of explicit BEGIN, COMMIT, and ROLLBACK TRANSACTION statements) and ensures that a procedure does not have to block or context switch midway through to wait for commit.

Finally, as we are building a commercial database product, we must take security into account in all aspects of the design. We were particularly concerned about the possibility of a "C injection attack" in which a malicious user might include C code in a T-SQL identifier (e.g., a table, column, procedure, or variable name) or string literal in an attempt to induce the code generator to copy this code into the generated code. Clearly, we cannot allow the execution of arbitrary C code by non-administrative users. To ensure that such an attack is not possible, we never include any user identifier names or data in the generated code even as comments and we convert string literals to a binary representation.

3.4 Query Interop

Compiled stored procedures do have limitations in the current implementation. The available query surface area is not yet complete and it is not possible to access regular tables from a compiled stored procedure. Recognizing these limitations, we implemented an additional mechanism that enables the interpreted query execution engine to access memory optimized tables. As noted in Section 3.1, we refer to this capability as interop. Interop enables several important scenarios including data import and export, ad hoc queries, support for query functionality not available in compiled stored procedures (including queries and transactions that access both regular and Hekaton tables).

4 Performance

We measured the benefits of native compilation through a set of simple experiments where we compare the number of instructions executed by the interpreted query execution engine with the number of instructions executed by an equivalent compiled stored procedure.

For the first experiment, we isolated and compared the cost of executing the simple predicate

```
Item = ? and Manufacturer = ? and Price > ?
```

using both the interpreted expression evaluator and the equivalent native code. We measured the cost of this predicate when a) evaluated against data stored in a regular disk-based B-tree table, b) using interop to access data stored in a Hekaton table, and c) using a compiled stored procedure to access the same Hekaton table. We ran these experiments without a suitable index as our intent was to measure the cost of the predicate evaluation not the cost of an index lookup. Since short circuiting can impact the cost of evaluating the predicate, we also measured the best (first column does not match) and worst (all columns match) case costs. The results are shown in Table 1. We use the most expensive case (the interpreted expression evaluator against a regular table with a row with all columns matching) as the baseline and report all other results as a percentage of this baseline.

Not surprisingly the compiled code is much more efficient (up to 10x fewer instructions executed) than the interpreted code. This improvement reflects the benefits of generating code where the data types and operations to be performed are known at compile time.

For our second experiment, we ran the simple queries shown in Figure 5. The only difference between these queries is that the first outputs the results to the client while the second saves the result in a local variable. Because outputting to the client incurs relatively high overhead, the first query is considerably more expensive than

	Interpreted expression (regular table)	Interop (Hekaton table)	Compiled (Hekaton table)
Worst case (all columns match)	100%	47%	11%
Best case (first column does not match)	43%	16%	4%

Table 1: Relative instruction cost for evaluating a predicate

the second. We ran the same three tests as for the first experiment comparing the interpreted query execution engine against a regular table, interop against a Hekaton table, and a compiled stored procedure against a Hekaton table. For this experiment we created an index on the Item and Manufacturer columns to simulate a more realistic scenario. The results are shown in Table 2. Once again we use the most expensive case (the interpreted query execution engine against a regular table with the results output to the client) as the baseline.

	Interpreted expression (regular table)	Interop (Hekaton table)	Compiled (Hekaton table)
Output to client	100%	66%	19%
Output to variable	94%	61%	6%

Table 2: Relative instruction cost for evaluating a query.

Once again, we see that the compiled code is much more efficient (up to 15x fewer instructions executed) than the interpreted code. As in the first experiment, many of the gains come from compile time knowledge of the data types and operations. We are also able to eliminate many virtual function calls and conditional branches whose outcomes are known at compile time.

```
--  Q1: Output to client
SELECT CustId FROM Sales
WHERE Item = ?
  AND Manufacturer = ?
  AND Price > ?

--  Q2: Output to variable
SELECT @CustId = CustId FROM Sales
WHERE Item = ?
  AND Manufacturer = ?
  AND Price > ?
```

Figure 5: Sample queries used in experiments.

Finally, a comment regarding the cost of compilation. Hekaton's primary focus is on OLTP queries where we compile once and execute thousands or millions of times. Thus, compiler performance was not a primary focus of this project. Nonetheless, for the vast majority of compiled stored procedures, the impact of compilation is not noticeable to the user. Most compilations complete in under one second and in many cases the cost of existing steps such as query optimization remain a significant fraction of the total end-to-end procedure creation cost. Some extremely complex procedures take longer to compile with the bulk of the time spent in compiler optimization. While such scenarios are rare, in the event that compilation takes unacceptably long, disabling some or all compiler optimizations generally improves compilation costs to an acceptable level albeit at some loss of runtime performance.

5 Concluding Remarks

Hekaton is a new database engine targeted for OLTP workloads under development at Microsoft. It is optimized for large main memories and many-core processors. It is fully integrated into SQL Server, which allows customers to gradually convert their most performance-critical tables and applications to take advantage of the very substantial performance improvements offered by Hekaton.

Hekaton achieves its high performance and scalability by using very efficient latch-free data structures, multiversioning, a new optimistic concurrency control scheme, and by compiling T-SQL stored procedure into

efficient machine code. As evidenced by our experiments, the Hekaton compiler reduces the instruction cost for executing common queries by an order of magnitude or more.

References

[1] Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. ACM SIGMOD 1990: 102-111

[2] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling: Hekaton: SQL server's memory-optimized OLTP engine. ACM SIGMOD 2013: 1243-1254

[3] Per-Ake Larson, Mike Zwilling, Kevin Farlee: The Hekaton Memory-Optimized OLTP Engine. IEEE Data Eng. Bull. 36(2): 34-40 (2013)

[4] IBM SolidDB, http://www.ibm.com/software/data/soliddb

[5] Oracle TimesTen, http://www.oracle.com/technetwork/products/timesten/overview/index.html

[6] SAP In-Memory Computing, http://www.sap.com/solutions/technology/in-memorycomputing-platform/hana/overview/index.epx

[7] Sybase In-Memory Databases, http://www.sybase.com/manage/in-memory-databases

[8] VoltDB, http://voltdb.com

[9] Martin Grund, Jens Krger, Hasso Plattner, Alexander Zeier,Philippe Cudr-Mauroux, Samuel Madden: HYRISE - A Main Memory Hybrid Storage Engine. PVLDB 4(2): 105-116 (2010)

[10] Martin Grund, Philippe Cudr-Mauroux, Jens Krger, Samuel Madden, Hasso Plattner: An overview of HYRISE - a Main Memory Hybrid Storage Engine. IEEE Data Eng. Bull.35(1): 52-57 (2012)

[11] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, Daniel J. Abadi: H-store: a high-performance, distributed main memory transaction processing system. PVLDB 1(2): 1496-1499 (2008)

[12] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, Anastasia Ailamaki: Data-Oriented Transaction Execution. PVLDB 3(1): 928-939 (2010)

[13] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker: OLTP through the looking glass, and what we found there. SIGMOD 2008: 981-992

[14] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, Babak Falsafi: Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35

[15] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, Mike Zwilling: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309 (2011)

[16] The LLVM Compiler Infrastructure, http://llvm.org/

[17] Phoenix compiler framework, http://en.wikipedia.org/wiki/Phoenix_compiler_framework

[18] Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 4(9): 539-550 (2011)

Runtime Code Generation in Cloudera Impala

Skye Wanderman-Milne
skye@cloudera.com

Nong Li
nong@cloudera.com

Abstract

In this paper we discuss how runtime code generation can be used in SQL engines to achieve better query execution times. Code generation allows query-specific information known only at runtime, such as column types and expression operators, to be used in performance-critical functions as if they were available at compile time, yielding more efficient implementations. We present Cloudera Impala, an open-source, MPP database built for Hadoop, which uses code generation to achieve up to 5x speedups in query times.

1 Introduction

Cloudera Impala is an open-source MPP database built for the Hadoop ecosystem. Hadoop has proven to be a very effective system to store and process large amounts of data using HDFS and HBase as the storage managers and MapReduce as the processing framework. Impala is designed to combine the flexibility and scalability that is expected from Hadoop with the performance and SQL support offered by commercial MPP databases. Impala currently executes queries 10-100x faster than existing Hadoop solutions and comparably to commercial MPP databases [1], allowing end users to run interactive, exploratory analytics on big data.

Impala is built from ground up to take maximal advantage of modern hardware and the latest techniques for efficient query execution. Impala is designed for analytic workloads, rather than OLTP, meaning it's common to run complex, long-running, CPU-bound queries. Runtime code generation using LLVM [3] is one of the techniques we use extensively to improve execution times. LLVM is a compiler library and collection of related tools. Unlike traditional compilers that are implemented as stand-alone applications, LLVM is designed to be modular and reusable. It allows applications like Impala to perform JIT compilation within a running process, with the full benefits of a modern optimizer and the ability to generate machine code for a number of architectures, by exposing separate APIs for all steps of the compilation process.

Impala uses LLVM to generate fully-optimized query-specific functions at runtime, which offer better performance than general-purpose precompiled functions. This technique can improve execution times by 5x or more for representative workloads. In this paper we describe how this achieved. Section 2 discusses how runtime code generation can be used to produce faster functions. Section 3 describes how we implement the code generation. Section 4 describes the implications of code generation for user-defined functions (UDFs). Section 5 details our results, and we conclude in Section 6.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
void MaterializeTuple(char* tuple) {
  for (int i = 0; i < num_slots_; ++i)
{
    char* slot = tuple + offsets_[i];
    switch(types_[i]) {
      case BOOLEAN:
        *slot = ParseBoolean();
        break;
      case INT:
        *slot = ParseInt();
        break;
      case FLOAT: …
      case STRING: …
      // etc.
    }
  }
}
```

interpreted

```
void MaterializeTuple(char* tuple) {
  *(tuple + 0) = ParseInt();    // i = 0
  *(tuple + 4) = ParseBoolean();// i = 1
  *(tuple + 5) = ParseInt();    // i = 2
}
```

codegen'd

Figure 1: Example function illustrating runtime optimizations possible with code generation

2 Advantages of runtime code generation

Impala uses runtime code generation to produce query-specific versions of functions that are critical to performance. In particular, code generation is applied to "inner loop" functions, i.e., those that are executed many times in a given query, and thus constitute a large portion of the total time the query takes to execute. For example, a function used to parse a record in a data file into Impala's in-memory tuple format must be called for every record in every data file scanned. For queries scanning large tables, this could be trillions of records or more. This function must therefore be extremely efficient for good query performance, and even removing a few instructions from the function's execution can result in large query speedups.

Without code generation, inefficiencies in function execution are almost always necessary in order to handle runtime information not known at compile time. For example, a record-parsing function that only handles integer types will be faster at parsing an integer-only file than a function that handles other data types such as strings and floating-point numbers as well. However, the schemas of the files to be scanned are unknown at compile time, and so the most general-purpose function must be used, even if at runtime it is known that more limited functionality is sufficient.

Code generation improves execution by allowing for runtime variables to be used in performance-critical functions as if they were available at compile time. The generated function omits the overhead needed to interpret runtime-constant variables. Figure 1 gives an example of this. The figure illustrates the advantages of using code generation on an example record-parsing function MaterializeTuple() (as in the example given above, the function parses a record and materializes the result into an in-memory tuple). On the left-hand side of the figure is pseudocode for the interpreted function, i.e., the function that is implemented without code generation. On the right-hand side is pseudocode for a possible code-generated version of the same function. Note that the interpreted function is appropriate for any query, since it makes no assumptions about runtime information, whereas the code-generated function is specific to a certain resolution of runtime information. The code-generated MaterializeTuple() function for a different query could be different.

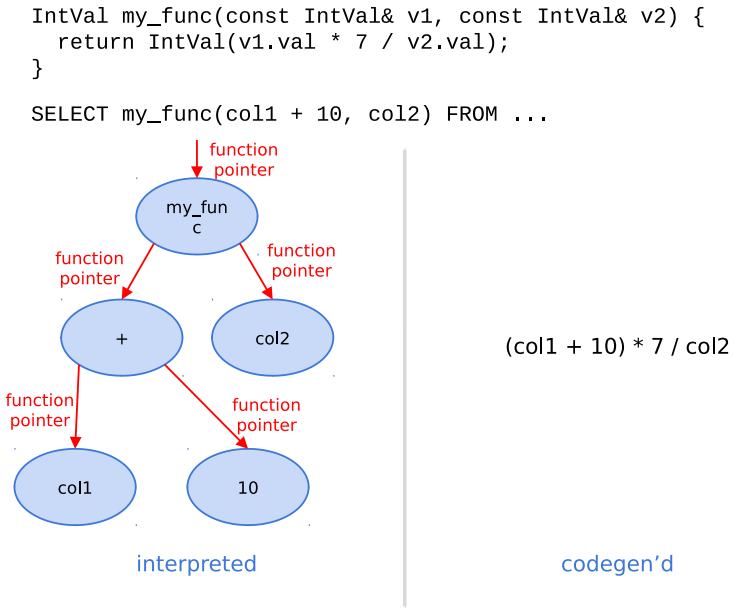


Figure 2: Expression tree optimization

More specifically, code generation allows us to optimize functions via the following techniques:

Removing conditionals: Runtime information that must be handled via an if or switch statement in an interpreted function can be resolved to a single case in the code-generated version, since the value of conditional is known at runtime. This is one of the most effective runtime optimizations, since branch instructions in the final machine code hinder instruction pipelining and instruction-level parallelism. By unrolling the for loop (since we know the number of iterations at runtime) and resolving the types, the branch instructions can be removed altogether.

Removing loads: Loading values from memory can be an expensive and pipeline-blocking operation. If the result of a load varies each time the function is invoked (e.g., loading the value of a slot in a tuple), there is nothing that can be done. However, if we know the load will always yield the same value on every invocation of the function, we can use code generation to substitute the load for the value. For example, in Figure 1, the offsets_ and types_ arrays are generated at the beginning of the query and do not vary, i.e., they are query-constant. Thus, in the code-generated version of the function, the values of these arrays can be directly inlined after unrolling the for loop.

Inlining virtual function calls: Virtual function calls incur a large performance penalty, especially when the called function is very simple, as the calls cannot be inlined. If the type of the object instance is known at runtime, we can use code generation to replace the virtual function call with a call directly to the correct function, which can then be inlined. This is especially valuable when evaluating expression trees. In Impala (as in many systems), expressions are composed of a tree of individual operators and functions, as illustrated in the left-hand side of Figure 2. Each type of expression that can appear in a tree is implemented by overriding a virtual function in the expression base class, which recursively calls its children expressions. Many of these expression functions are quite simple, e.g., adding two numbers. Thus, the cost of calling the virtual function often far exceeds the cost of actually evaluating the function. As illustrated in Figure 2, by resolving the virtual function calls with code generation and then inlining the resulting function calls, the expression tree can be evaluated directly with no function call overhead. In addition, inlining functions increases instruction-level parallelism, and allows the

compiler to make further optimizations such as subexpression elimination across expressions.

3 Generating code with LLVM

When a query plan is received by Impala's backend execution engine, LLVM is used to generate and compile query-specific versions of performance-critical functions before the query execution begins. This section explains in detail how the functions are generated before being compiled.

3.1 LLVM IR

LLVM primarily uses an intermediate representation (IR) for all parts of code generation. LLVM IR [8] resembles assembly language, being composed of a number of simple instructions that often have direct mappings to machine code. Frontends to LLVM, such as the Clang C++ compiler [5], generate IR, which can then be optimized and lowered to machine code by LLVM. We use two techniques for generating IR functions in Impala: using LLVM's IRBuilder, which allows for programmatically generating IR instructions, and cross-compiling C++ functions to IR using Clang.

3.2 IRBuilder

LLVM includes an IRBuilder class [7] as part of their C++ API. The IRBuilder is used to programmatically generate IR instructions, and as such can be used to assemble a function instruction by instruction. This is akin to writing functions directly in assembly: it's very simple, but can be quite tedious. In Impala, the C++ code for generating an IR function using the IRBuilder is generally many times longer than the C++ code implementing the interpreted version of the same function. However, this technique can be used to construct any function.

3.3 Compilation to IR

Instead of using the IRBuilder to construct query-specific functions, we generally prefer to compile a C++ function to IR using Clang, then inject query-specific information into the function at runtime. This allows us to write functions in C++ rather than constructing them instruction by instruction using the IRBuilder. We also cross-compile the functions to both IR and native code, allowing us to easily run either the interpreted or code-generated version. This is useful for debugging: we can isolate whether a bug is due to code generation or the function itself, and the native functions can be debugged using gdb.

Currently, our only mechanism for modifying compiled IR function is to replace function calls to interpreted functions with calls to equivalent query-specific generated functions. This is how we remove virtual functions calls as described in Section 2. For example, we cross compile many of the virtual functions implementing each expression type to both IR and native code. When running with code generation disabled, the native functions are run as-is, using the interpreted general-purpose expression implementations. When code generation is enabled, we recursively find all calls to child expressions and replace them with calls to code-generated functions.

Of course, this technique alone doesn't allow us to take full advantage of code generation. It doesn't help us with many of the techniques described in section 2, such as removing conditionals and loads. For now we generate functions that benefit from these techniques using the IRBuilder. However, we are currently developing a new framework for modifying precompiled IR functions. Returning to to the MaterializeTuple() example in Figure 1, the main optimizations we would like to perform on the interpreted code in order to take advantage of runtime information are (1) to unroll the for loop using the known num_slots_ variable, so we can replace each iteration with iteration-specific runtime information, and (2) to replace accesses of offsets_ and types_ with the actual values. Once we have a framework for these transformations, we will be able to implement code-generated functions more easily and quickly than we can with the IRBuilder.

Query	Code generation disabled	Code generation enabled	Speedup
select count(*) from lineitem	3.554 sec	2.976 sec	1.19x
select count(l_orderkey) from lineitem	6.582 sec	3.522 sec	1.87x
TPCH-Q1	37.852 sec	6.644 sec	5.70x

Table 3: Query times with and without code generation

4 User-defined functions

Impala provides a C++ user-defined function (UDF) API, and the most conventional method for authoring a UDF is to implement it in C++ and compile it to a shared object, which is dynamically linked at runtime. However, as discussed above, Impala can compile and execute IR functions generated by Clang. We take advantage of this functionality to execute UDFs compiled to IR, rather than to a shared object. This enables inlining function calls across user functions, meaning UDFs can have identical performance to Impala's built-ins.

In addition to performance benefits, this architecture easily allows UDFs to be authored in other languages. Just as we use Clang to compile C++ functions to IR, any language with an LLVM frontend can be used to author UDFs without modification to the query engine. For example, Numba [9] allows compilation from Python to IR. Using our approach, a developer could author UDFs in Python that would be even more performant than C++ UDFs statically compiled to shared objects.

5 Experimental results

In Table 3, we show the effectiveness of runtime code generation as we increase the complexity of the query. These queries were run on a 10-node cluster over a 600M-row Avro [4] data set. In the first query, we do a simple count of the rows in the table. This doesn't require actually parsing any Avro data, so the only benefit of code generation is to improve the efficiency of the count aggregation, which is already quite simple. The resulting speedup from code generation is thus quite small. In the second query, we do a count of a single column, which requires parsing the Avro data in order to detect null values. This increases the benefit of code generation by 60% over the simpler query. Finally, we run the TPCH-Q1 query, which is reproduced in Figure 3. This involves multiple aggregates, expressions, and group by clauses, resulting in a much larger speedup.

```
select
  l_returnflag, l_linestatus, sum(l_quantity), sum(l_extendedprice),
  sum(l_extendedprice * (1 - l_discount)),
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
  avg(l_quantity), avg(l_extendedprice), avg(l_discount), count(1)
from lineitem
where l_shipdate<='1998-09-02'
group by l_returnflag, l_linestatus
```

Figure 3: TPCH-Q1 query

In Table 4, we look at how runtime code generation reduces the number of instructions executed when running TPCH-Q1. These counts were collected from the hardware counters using the Linux perf tool. The counters are collected for the entire duration of the query and include code paths that do not benefit from code generation.

	# Instructions	# Branches
Code generation disabled	72,898,837,871	14,452,783,201
Code generation enabled	19,372,467,372	3,318,983,319
Speedup	4.29x	3.76x

Table 4: Instruction and branch counts for TPCH-Q1

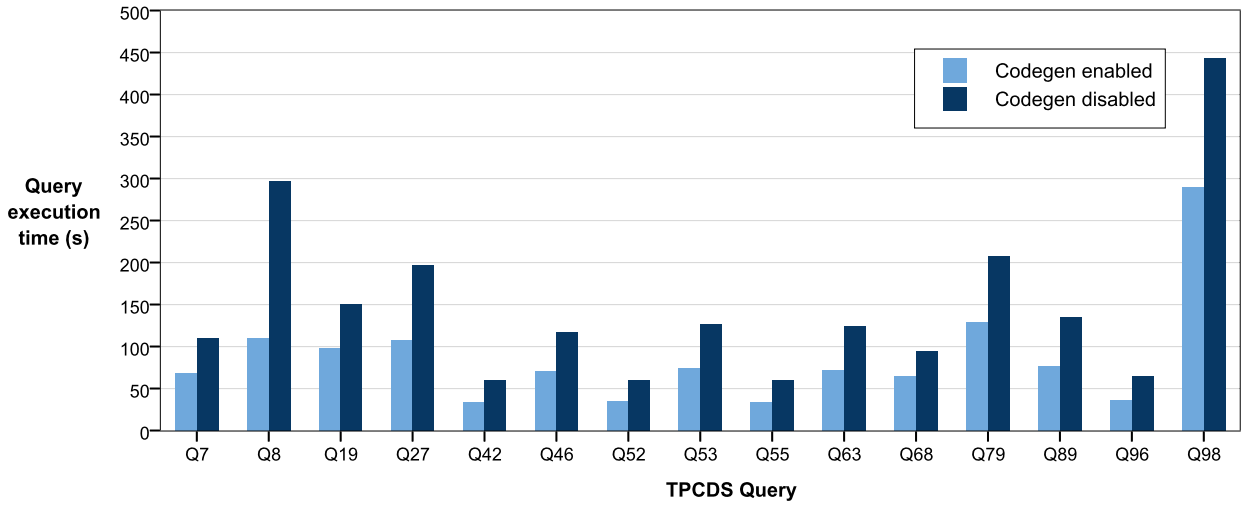


Figure 4: TPC-DS query execution with and without code generation

Finally, in Figure 4, we present the results of running several slightly-modified TPC-DS queries with and without code generation. These queries were run on a 10-node cluster over a 1-terabyte scale factor Parquet [6, 2] dataset. These queries do not achieve the 5x speedup seen on the above TPCH-Q1 result due to some sections of the query execution not implementing code generation. In particular, nearly all the TPC-DS queries contain an ORDER BY clause, which does not currently benefit from code generation, and the Parquet file parser does not currently use code generation (we used Parquet rather than Avro, which does use code generation, for these results because otherwise the queries are IO-bound). However, there is still a significant speedup gained on every query, and we expect this to become much larger once more of the execution can be code generated.

6 Conclusions and future work

LLVM allows us to implement a general-purpose SQL engine where individual queries perform as if we had written a dedicated application specifically for that query. Code generation has been available in Impala since the initial release, and we're excited about the ideas we have to further improve it. We currently only code generate functions for certain operators and functions, and are expanding our efforts to more of the execution. The more of the query that is generated, the more we can take advantage of function inlining and subsequent inter-function optimizations. Eventually we would like the entire query execution tree to be collapsed into a single function, in order to eliminate almost all in-memory accesses and keep state in registers.

We are also working on a project integrating a Python development environment with Impala, taking advantage of Impala's ability to run IR UDFs. The project will allow users, working completely within the Python shell or scripts, to author UDFs and run them across a Hadoop cluster, with parameters and results being auto-

matically converted between native Python and Impala types as needed. We hope this will enable new uses cases involving scientific computing for Impala and Hadoop.

These are just a few of the many possible directions we can take with runtime code generation. The technique has proven enormously useful, and we imagine it will become more widespread in performance-critical applications.

References

[1] Erickson, Justin, Greg Rahn, Marcel Kornacker, and Yanpei Chen. *"Impala Performance Update: Now Reaching DBMS-Class Speed."* Cloudera Developer Blog. Cloudera, 13 Jan. 2014. http://blog.cloudera.com/blog/2014/01/impala-performance-dbms-class-speed/

[2] *"Introducing Parquet: Efficient Columnar Storage for Apache Hadoop."* Cloudera Developer Blog. Cloudera, 13 March 2013. http://blog.cloudera.com/blog/2013/03/introducing-parquet--columnar-storage-for-apache-hadoop/

[3] "LLVM: An Infrastructure for Multi-Stage Optimization", Chris Lattner. *Masters Thesis*, Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[4] http://avro.apache.org/

[5] http://clang.llvm.org/

[6] http://parquet.io/

[7] http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html

[8] http://llvm.org/docs/LangRef.html

[9] http://numba.pydata.org

Database Application Developer Tools Using Static Analysis and Dynamic Profiling

Surajit Chaudhuri, Vivek Narasayya, Manoj Syamala

Microsoft Research
{surajitc,viveknar,manojsy}@microsoft.com

Abstract

Database application developers use data access APIs such as ODBC, JDBC and ADO.NET to execute SQL queries. Although modern program analysis and code profilers are extensively used during application development, there is a significant gap in these technologies for database applications because these tools have little or no understanding of data access APIs or the database system. In our project at Microsoft Research, we have developed tools that: (a) Enhance traditional static analysis of programs by leveraging understanding of database APIs to help developers identify security, correctness and performance problems early in the application development lifecycle. (b) Extend the existing DBMS and application profiling infrastructure to enable correlation of application events with DBMS events. This allows profiling across application, data access and DBMS layers thereby enabling a rich class of analysis, tuning and profiling tasks that are otherwise not easily possible.

1 Introduction

Relational database management systems (DBMSs) serve as the backend for many of today's applications. Such *database applications* are often written in popular programming languages such as C++, C# and Java. When the application needs to access data residing in a relational database server, developers typically use data access APIs such as ODBC, JDBC and ADO.NET for executing SQL statements and consuming the results. Application developers often rely on integrated development environments such as Microsoft Visual Studio or Eclipse which provide a variety of powerful tools to help develop, analyze and profile their applications. However, these development environments have historically had limited understanding of the interactions between the application and the DBMS. Thus a large number of security, correctness and performance issues can go undetected during the development phase of the application, potentially leading to high cost once the application goes into production.

In this paper, we summarize the key ideas, techniques and results of our project at Microsoft Research to develop tools for database application developers that leverage static program analysis [3] as well as dynamic profiling [2] of the application at runtime. We first discuss the framework for statically analyzing database application binaries to automatically identify security, correctness and performance problems in the database application. Our idea is to adapt data and control flow analysis techniques of traditional optimizing compilers

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

by exploiting semantics of data access APIs and the database domain to provide a set of analysis services on top of the existing compiler. These services include: (a) Extracting the set of SQL statements that can execute in the application. (b) Identifying properties of the SQL statements such as tables and columns referenced. (c) Extracting parameters used in the queries and their binding to program variables. (d) Extracting properties of how the SQL statement results are used in the application. (e) Analyzing user input and their propagation to SQL statements. Using the above services, we have built vertical tools for: detecting SQL injection vulnerability, extracting the SQL workload from application binary, identifying opportunities for SQL query performance optimizations, and identifying potential data integrity violations.

While such static analysis can greatly aid database application developers, many issues in the application can only be detected at runtime. For example, if there is a deadlock in the DBMS, detecting which tasks in the application are responsible for causing that deadlock today can be non-trivial despite the availablity of both application and database profiling tools. The key reason why such tasks are challenging is that the *context of an application* (threads, functions, loops, number of rows from a SQL query actually consumed by the application, etc.) and the *context of the database server* when executing a statement (duration of execution, deadlocks, duration for which the statement was blocked, number of rows returned etc.) cannot be easily correlated with each other. We discuss an infrastructure that can obtain and correlate the appropriate application context with the database context, thereby enabling a class of development, debugging and tuning tasks that are today difficult to achieve for application developers. We conclude with a discussion of some open issues and future work.

2 Motivating Scenarios

We discuss a set of motivating scenarios around security, correctness and performance of database applications that can potentially be detected via static analysis and/or dynamic profiling.

SQL injection vulnerability: A well known example of such a security problem is SQL injection vulnerability. Applications that execute SQL queries based on user input are at risk of being compromised by malicious users who can inject SQL code as part of the user input to gain access to information that they should not. Detecting SQL injection vulnerability at application development time can help developers correct the problem even before the application is deployed into production, thereby avoiding expensive or high profile attacks in production (e.g. [4]).

Database integrity constraint violations: In many real-world applications, certain database integrity constraints are enforced in the application layer and not the database layer. This is done for performance reasons or to avoid operational disruption to an application that has already been deployed. For example, in a hosted web service scenario the DBA might be reluctant to pay the cost of altering an existing table of a deployed application. In such scenarios, given a database constraint such as $[Products].[Price] > 0$ as input, it would be useful if we could automatically identify all places in the application code where the *Price* column can potentially be updated, and add an assertion in the application code to verify whether the constraint is honored.

Enforcing best practices: There can be correctness or performance problems due to the way the queries are constructed or used in the application. For example, there can be mismatch between the data type used in the application (e.g. *int*) and the data type of the column in the database (*smallint*). Such a mismatch is not detected by todays application development tools, which can lead to unexpected application behavior at runtime. Development teams need automatic support for enforcing a set of best practices in coding (e.g. - no SELECT * queries or always use the ExecuteScalar() ADO.NET API for queries that return scalar values). While code analysis tools like FxCop [5] aid to an extent for best practices, these tools have no database and data access specific domain knowledge to support the above kind of analysis.

Root-causing deadlocks in the DBMS: Certain database application issues can only be detected at runtime. Consider an application that executes two concurrent tasks each on behalf of a different user. Each task invokes certain functions that in turn issue SQL statements that read from and write to a particular table in the database.

Suppose a bug causes SQL statements issued by the application to deadlock with one another on the server. The database server will detect the deadlock and terminate one of the statements and unblock the other. This is manifested in the application as one task receiving an error from the server and the other task running to completion normally. Thus, while it is possible for the developer to know that there was a deadlock (by examining the DBMS profiler output or the server error message of the first task) it is difficult for the developer to know which function from the other task issued the corresponding statement that caused the server deadlock. In general, having the ability to identify the application code that is responsible for the problem in the database server can save considerable debugging effort for the developer.

Suggesting query hints: By observing usage of data access APIs of the application, it may be possible to improve performance via use of query hints. For example, it is common for applications to execute a query that returns many result rows but not actually consume all the results, e.g. because user actions drive how many results are viewed. In most database systems, by default the query optimizer generates a plan that is optimized for the case when all N rows in the result set are needed. If only the top k rows are required instead ($k < N$), the optimizer can often generate a much more efficient plan. Such a plan can be generated by providing a "FAST k" query hint. The important point to note is that the information about what value of k (number of rows consumed by the application) is appropriate is available only by profiling the application context.

3 Solution Overview

The tool can operate in two modes. In the static analysis mode (see Figure 1) the tool performs custom static analysis on the input binary as explained in [3]. The output is a set of potential security, performance and correctness problems in the application pertaining to use of the database. In the dynamic analysis mode (see Figure 2) the tool instruments the application binary and converts it into an event provider. Once instrumented, the developer launches the application after turning on tracing for all the three event providers: (1) Microsoft SQL Server tracing (2) ADO.NET tracing and (3) Instrumented events from the application. This allows events containing both application context and database context to be logged into the ETW [6] event log. The key post-processing step is done by our Log Analyzer that correlates application and server events using a set of matching techniques as explained in paper [2].

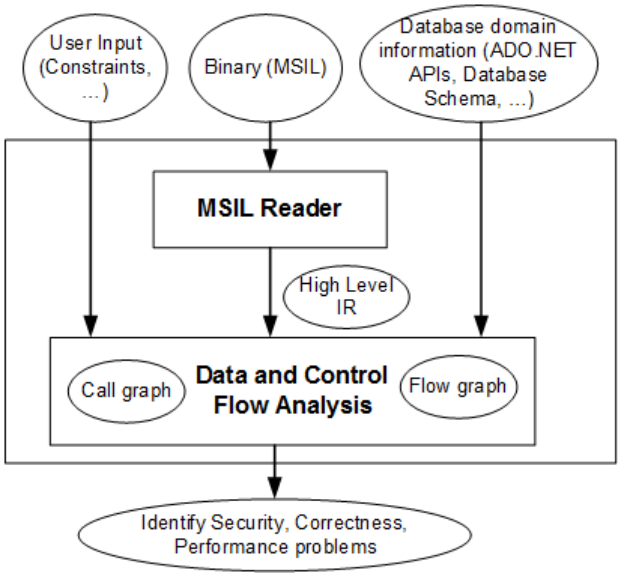


Figure 1: Overview of architecture of static analysis tool.

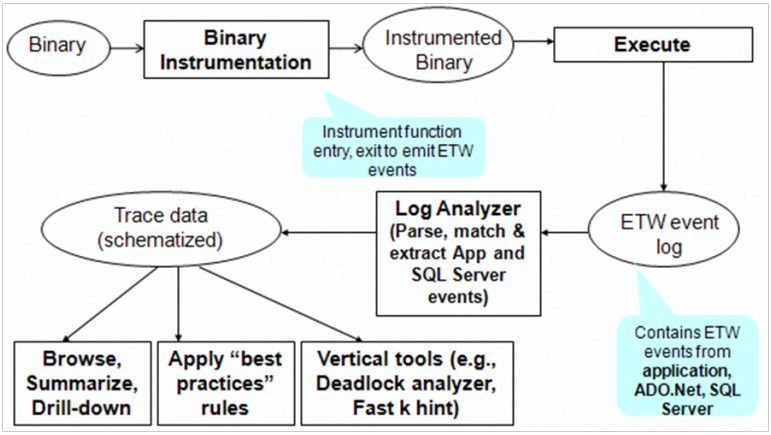


Figure 2: Overview of architecture of dynamic profiling tool.

Static Analysis: Our implementation of the static analysis tool relies on the Phoenix compiler framework [1]. We rely upon Phoenix to: (1) Convert the application binary in Microsoft Intermediate Language (MSIL) into an intermediate representation (IR) that our analysis operates upon. (2) Iterate over function unit(s) within the binary. (3) Provide the flow graph in order to iterate over basic blocks within a function unit. (4) Iterate over individual instructions in the IR within a basic block. (5) Provide extensions to dynamically extend the framework types like function units and basic blocks. (6) Provide a call graph that represents the control flow across function units and in case of dynamic analysis. (7) Instrument the binary so that it turns into a provider of ETW events. We extend this framework to develop the following primitives.

- *Extract SQL:* Given a function in the program binary, this primitive returns a set of SQL statement handles. A handle is a unique identifier that is a (*line number*, *ordinal*) pair in that function. It represents a SQL statement that can execute at that line number.

- *Identify SQL properties:* Given a *handle* to a SQL statement, this primitive returns properties of the SQL statement such as the SQL string, number and database types of columns in the result of the SQL statement, tables and columns referenced in the statement, and optimizer estimated cost.

- *Extract Parameters:*. Given a *handle* to a SQL statement this primitive returns the parameters of the statement along with the program variable/expression that is bound to that parameter, and its data type in the application.

- *Extract Result Usage:* Given a *handle* to a SQL statement, this primitive returns properties of how the result set is consumed in the application. In particular, it returns each column in the result set that is bound to a variable in the program, along with the type of the bound program variable.

- *Analyze User Input:* Given a *handle* to a SQL statement this primitive identifies all user inputs in the program such that the user input value v satisfies a contributes to relationship to the SQL string of the statement. A contributes to relationship is defined as either: (a) v is concatenated into the SQL string. (b) v is passed into a function whose results are concatenated into the SQL string.

The "vertical" functionality such as identifying SQL injection vulnerabilities, workload extraction and detecting potential data integrity violations are built using the above primitives. For additional technical details we refer the reader to [3].

Dynamic Profiling: Once the binary has been instrumented, the developer can click through a wizard (exposed as an Add-In to Microsoft Visual Studio), which launches the application after turning on tracing

for all the three event providers: (1) Microsoft SQL Server tracing (2) ADO.net tracing and (3) Instrumented events from the application. This allows events containing both application context and database context to be logged into the ETW event log. The key post-processing step is done by our Log Analyzer module that correlates application and server events using a set of matching techniques [2]. This matching is non-trivial since today there is no unique mechanism understood both by ADO.Net and Microsoft SQL Server to correlate an application event with a server event. The above collection and matching enables us to bridge the two contexts and provide significant added value to database application developers via "verticals" such as root causing deadlocks and fast k query hints.

4 Evaluation

We first provide a few examples using screenshots showing the functionality of the tools, and then summarize the results of applying the static analysis tool on a couple of real-world applications. A screenshot of the output of static analysis by the tool is shown in Figure 3. The left hand pane shows the functions in the binary. The SQL Information grid shows the SQL string, the SQL injection status (UNSAFE in this example). It also shows the actual line number in the code where the user input (leading to this vulnerability) originated, and the line number where the SQL statement is executed.

Figure 4 shows another screenshot of the static analysis tool. In this scenario, the user specifies via input that they expect the database constraint $[Products].[Price] > 0$ to hold, where Products is a table and Price is a column in that table. The right pane displays: (1) The fully formed SQL statement and the line number in the application where the SQL can execute when the application is run (in this case an INSERT statement). (2) Information about the parameters that are bound to the SQL statement. These include the parameter name, the data type and the application variable that is bound to the SQL parameter. (3) The application constraint corresponding to the input data integrity constraint specified by the user and the line number where it should be added. In this example the constraints analysis pane shows that expression ($price1 > 0$), where $price1$ is an application variable, will enforce the database constraint $[Products].[Price] > 0$ if it is placed at line number 279 in the application code.

In Figure 5 we see a screenshot showing the result of the dynamic profiling. The output of the tool is the summary/detail view as shown in the figure. Developers can get a summary and detail view involving various counters from the application, ADO.NET and Microsoft SQL Server, navigate the call graph hierarchy and invoke specific verticals. The Summary view gives the function name, aggregate time spend in a function, how many times the function was invoked and aggregate time spend executing the SQL statement (issued by the particular function) in the database server. Today the Function, Exclusive Time and Number of Invocations counters can be obtained from profiling the application using application side profiling tools such as Visual Studio Profiler; however the SQL Duration is an example of our value-add since it merges in database context into the application context. Consider the function *ReadStuff* which issues a SQL call. From the Summary view the developer can determine that the function was called twice and the aggregate time it spend inside this function (across all instances) was 5019 *msec*. Out of the total time spend in the function, most of the time was spend executing SQL (5006 *msec*). The Detail view gives more information at a function instance level. The tool allows drill down to display attributes of all the statements that were issued under the particular instance of the function or statements that were issued under the call tree of the particular instance of the function. The attributes of the SQL statement that are displayed include counters like duration, reads, writes, and also data access counters like reads issued by the application, and the data access API type, corresponding to the SQL that was issued.

A sample output from the deadlock analysis vertical is shown in Figure 6. The Microsoft SQL Server Profiler trace produces a Deadlock Event which contains the wait-for graph that describes a deadlock. The graph contains the statements being executed that resulted in the deadlock as well as timestamp, and client

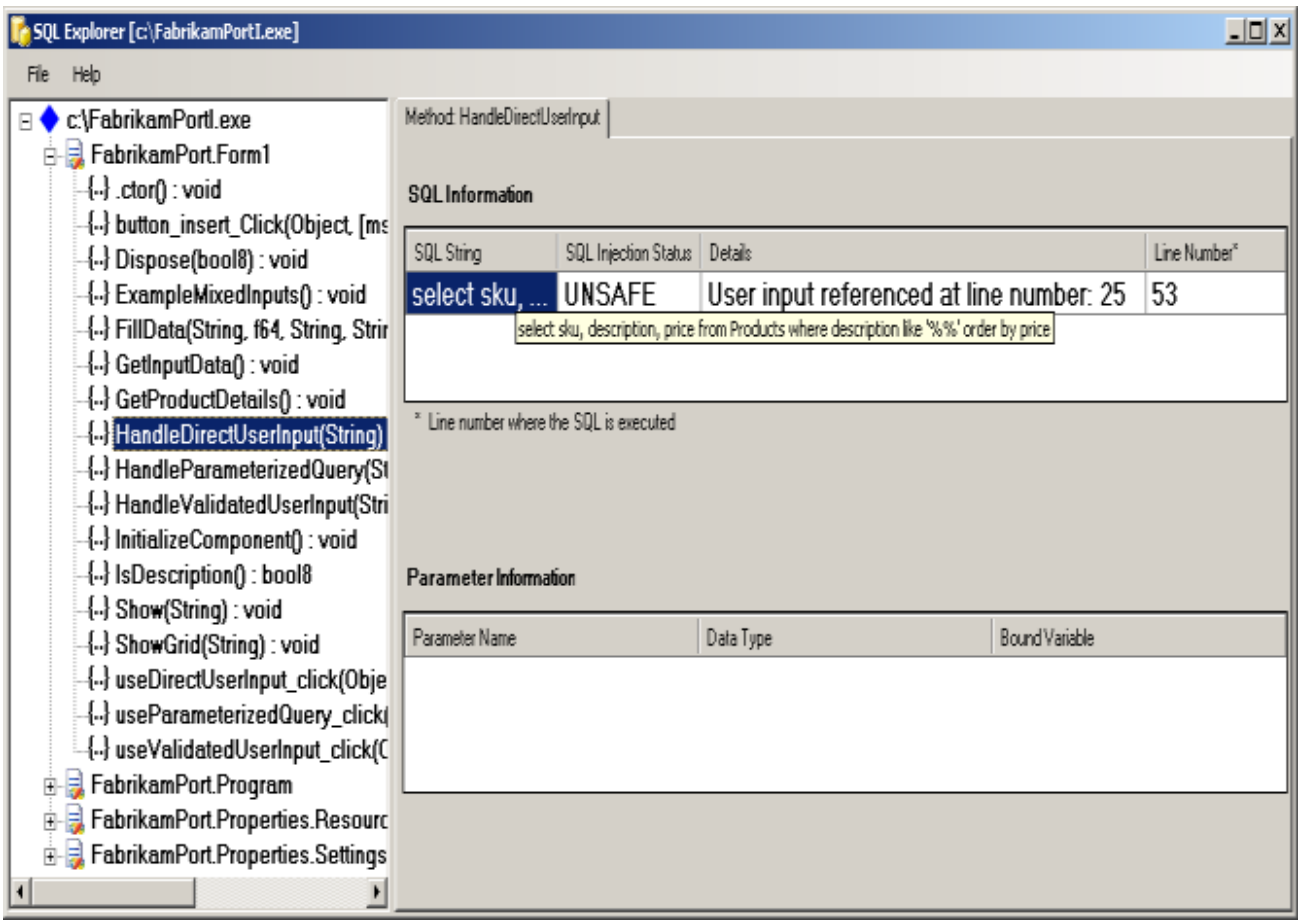


Figure 3: Output of the tool for SQL injection detection.

process id(s) information. The log analyzer extracts this information and stores it in the schematized application trace under the root node of the tree (as an event of type deadlock). For each such deadlock event, the deadlock analysis vertical finds the statements issued by the application that correspond to the statements in the deadlock event. Note that once we find the statement, we get all its associated application context such as function and thread. This can then be highlighted to the developer so they can see exactly which functions in their application issues the statements that lead to the deadlock.

We report briefly our experiences of running our static analysis tools on a few real world database applications: Microsofts Conference Management Toolkit (CMT): CMT [10] is a web application sponsored by Microsoft Research that handles workflow for an academic conference. SearchTogether [11]: An application that allow multiple users to collaborate on web search. For each application we report our evaluation of the Workload Extraction vertical. Our methodology is to compare the workload extracted by our tool with the workload obtained by manual inspection of the application code. The summary of results is shown in Table 5. The column *Total Num. of SQL statements* reports the number of that SQL statements that we were able to manually identify by examining the source code of the application. The column *Num. of SQL statements extracted* refers to the number of statements that were extracted by our static analysis tool. Along with the SQL statements we were able to extract parameter information as well. Thus, even though the actual parameter values are not known at compile time, we are able to extract syntactically valid queries. Thus it is possible, for example, to obtain a query execution plan for such queries. CMT and SearchTogether applications both mostly use parameterized stored procedures.

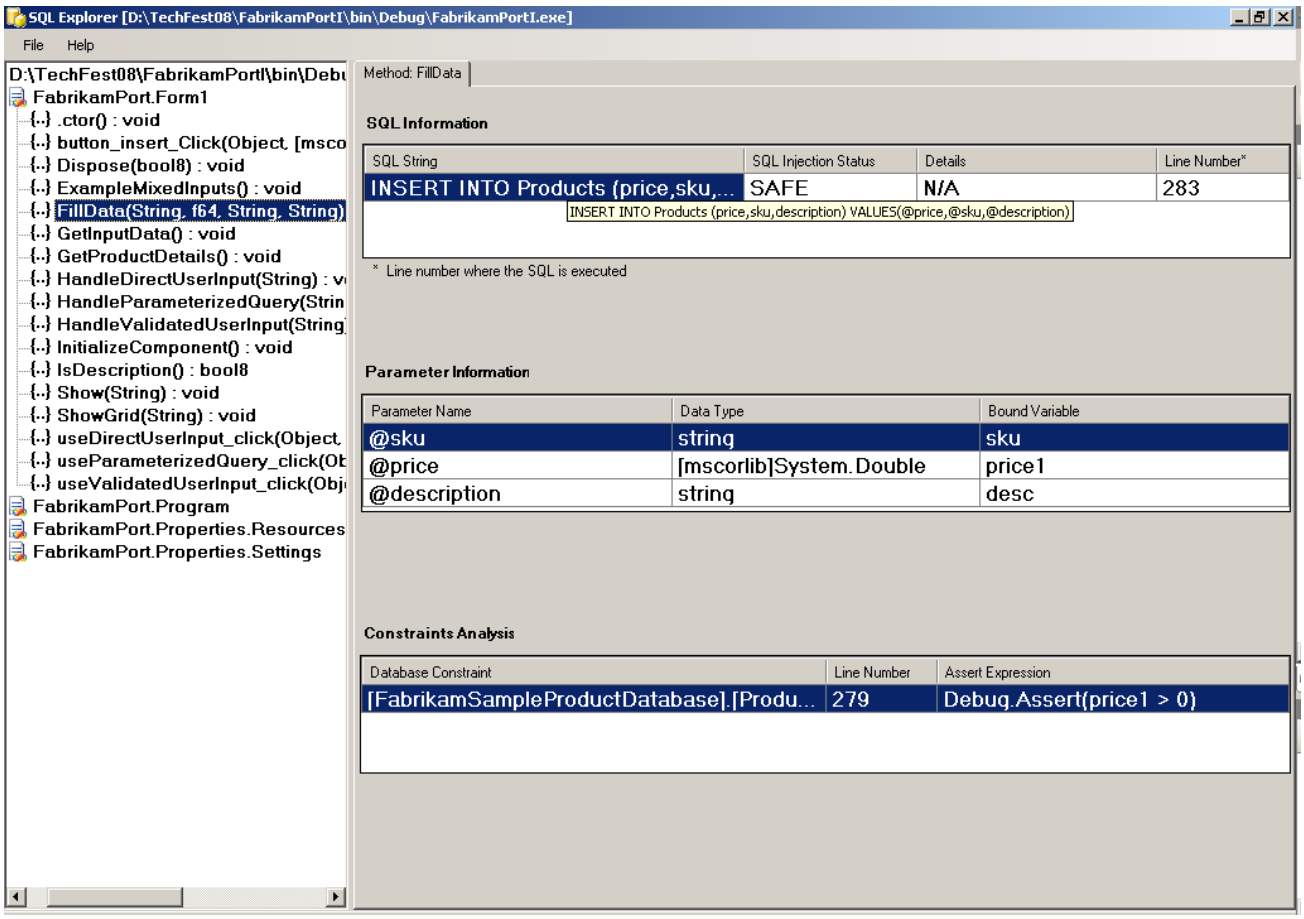


Figure 4: Detection of potential data integrity violation.

Application	Lines of code	Total Num. of SQL statements	Num. of SQL statements extracted
CMT	36,000+	621	350
SearchTogether	1,700	40	35

Table 5: Summary of results for workload extraction.

The cases where we were not able to extract SQL strings were due to the following reasons. First, there many ADO.NET APIs exposed by the providers that are used in these applications. Our current implementation does not cover the entire surface area of all the ADO.NET APIs. In some cases in SearchTogether, the *SQLCommand* object is a member variable of a class. The object is constructed in one method and referenced in another method. In this case, the global data flow analysis of our current implementation is not sufficient since the variable (the *SQLCommand* object in this case) is not passed across the two methods. Capturing this case requires tracking additional state of the *SQLCommand* object, which our current implementation does not. We also ran our SQL injection detection tool on all the three applications. We detected no SQL injection vulnerabilities in CMT and SearchTogether. In these applications user input is bound to parameters and executed as parameterized SQL.

5 Conclusion

In this paper we discussed how by exploiting our understanding of the semantics of data access APIs it is possible to detect a class of problems in the application through static program analysis. We also showed that the ability to

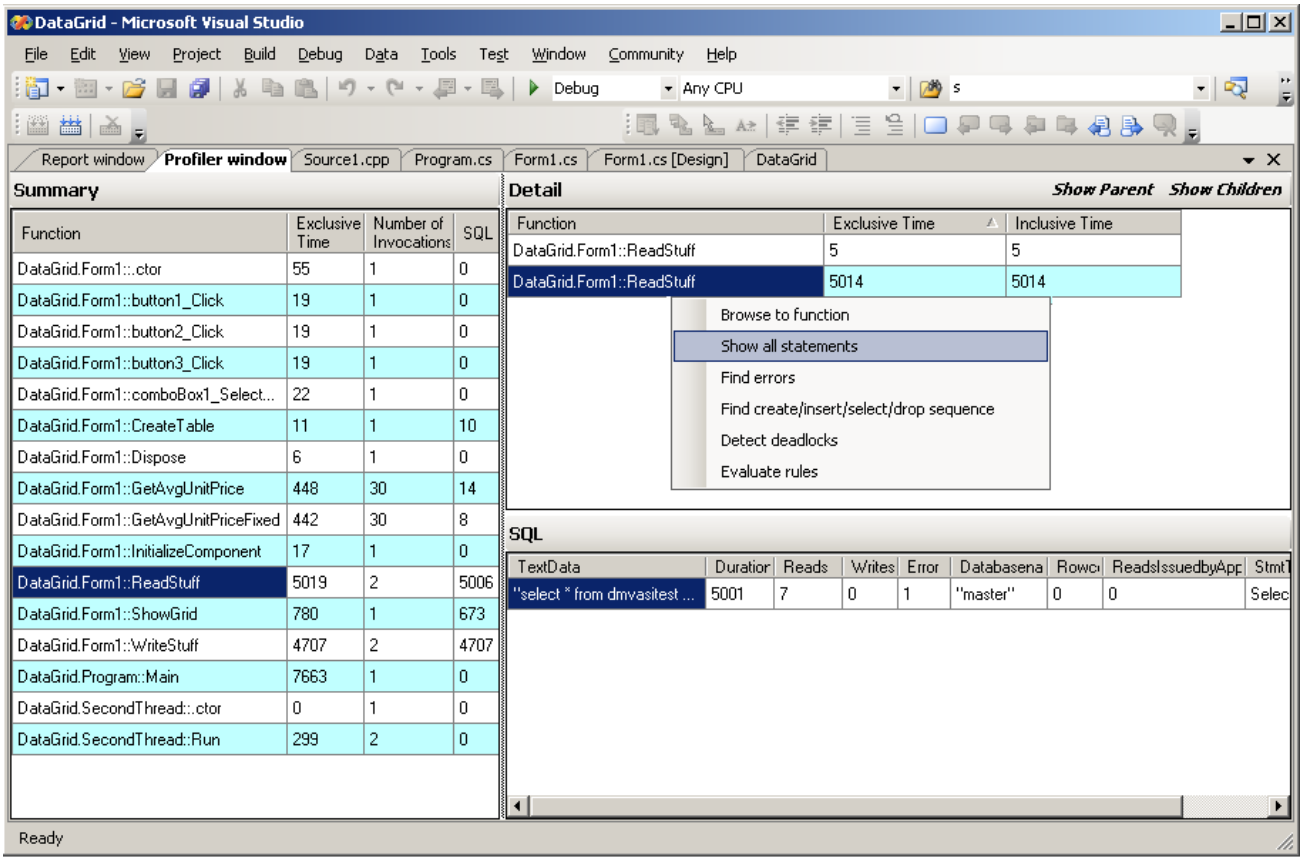


Figure 5: Summary and Detail views of Dynamic Profiling.

automatically profile and correlate application context and database context at runtime further enables expanding the class of issues that can be detected. We conclude with a discussion of a few open issues that are potential areas for future work.

- *Extending to applications in the cloud.* Applications running on cloud infrastructure often reference multiple services in addition to databases such as caching, queueing and storage. The applications use well defined APIs to interact with these services, similar to data access APIs for databases. Extending the techniques described here for such multi-tier applications by exploiting the semantics of these APIs could be valuable to application developers.

- *Analyzing SQL code.* Many database applications use stored procedures. Analyzing the SQL statements within these stored procedures using static analysis can help identify performance and security issues similar to those described here.

- *Profiling overhead.* Dynamic profilng can impose non-trivial overheads on performance both within the application and in the DBMS. While this may be acceptable during the application development phase, such overheads are unacceptable in a production setting. Thus optimizations that could limit the overheads of dynamic profiling could greatly broaden the scope of applicability of such tools to production use.

- *Correlating events in multi-tier applications.* Many database engines do not natively support propagation of a unique identifier that tracks a particular ”activity” (say for example a new order transaction in the application code) between the application and the database engine. Native support for such propagation

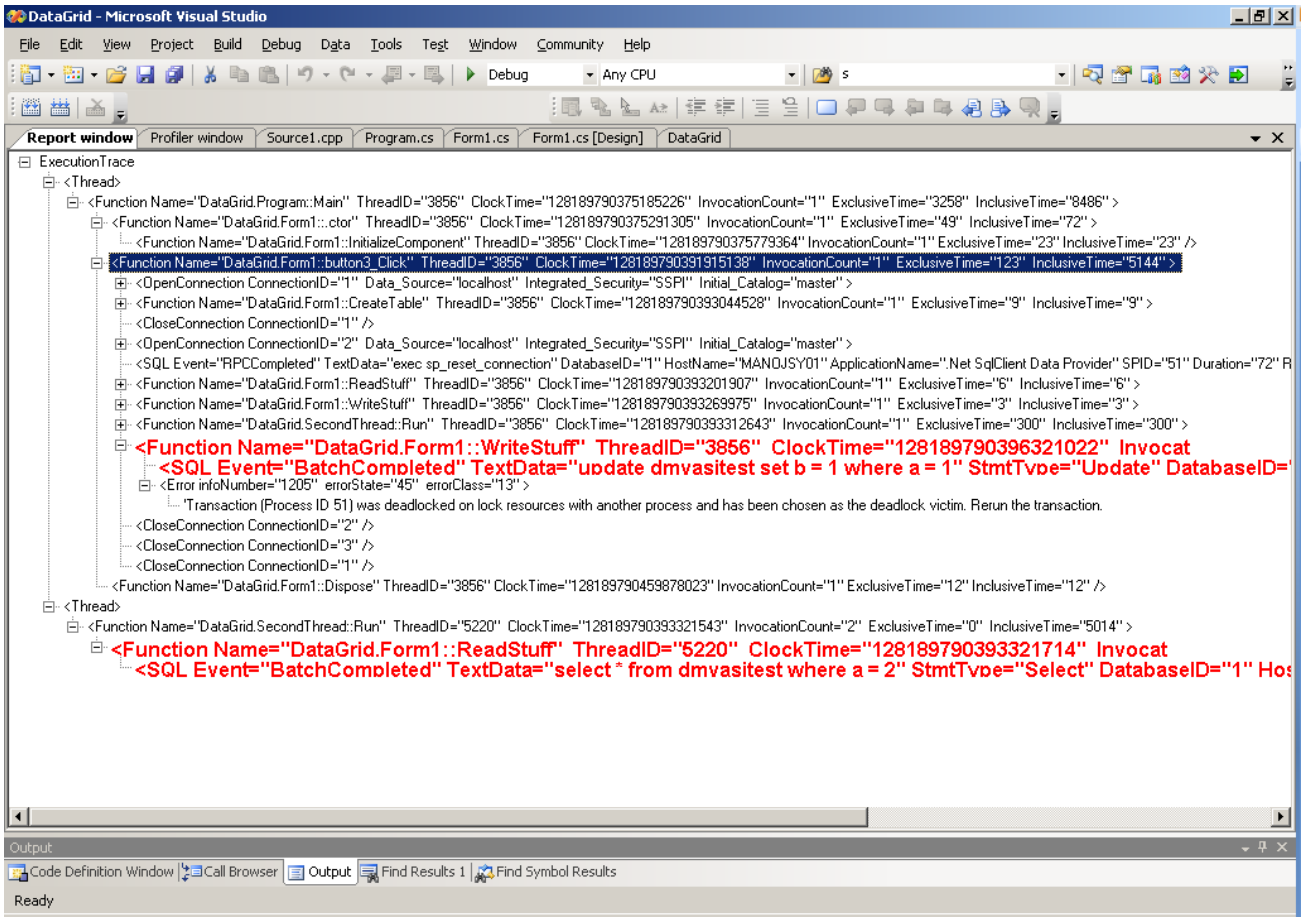


Figure 6: Output of deadlock analysis vertical.

of such an identifer would greatly ease the problem of correlating events logged at the application layer and events logged by the database server and eliminate the need to resort to approximate correlations such as time and textual similarity.

- *Automated performance optimizations*. Our focus has primarily been on *detecting* a range of security, performance and correctness problems in database applications so that developers can take appropriate actions to address the issues. An interesting related area that has received some attention more recently is automatically modifying application and/or database (SQL) code to improve program performance. Examples include automatically replacing imperative application code with equivalent declarative SQL code to improve efficiency [8] and automatically prefetching query results from the database based on understanding control flow in the application code [9].

References

[1] Phoenix compiler framework, http://en.wikipedia.org/wiki/Phoenix_compiler_framework

[2] Surajit Chaudhuri, Vivek R. Narasayya, Manoj Syamala: Bridging the Application and DBMS Profiling Divide for Database Application Developers. VLDB 2007: 1252-1262

[3] Arjun Dasgupta, Vivek R. Narasayya, Manoj Syamala: A Static Analysis Framework for Database Applications. ICDE 2009: 1403-1414

[4] United Nations vs. SQL Injections. http://hackademix.net/2007/08/12/united-nations-vs-sql-injections.

[5] FxCop: Application for analyzing managed code assemblies. http://msdn.microsoft.com.

[6] Event Tracing for Windows (ETW). http://msdn.microsoft.com.

[7] A. Aho, R. Sethi, and J. Ullman. Compilers. Principles, Techniques and Tools. Addison Wesley.

[8] Alvin Cheung, Owen Arden, Samuel Madden, Andrew C. Myers: Automatic Partitioning of Database Applications. PVLDB 5(11): 1471-1482 (2012).

[9] Karthik Ramachandra, S. Sudarshan: Holistic optimization by prefetching query results. SIGMOD Conference 2012: 133-144

[10] Microsoft Conference Management Service (CMT). http://msrcmt.research.microsoft.com/cmt/

[11] Microsoft SearchTogether application. http://research.microsoft.com/searchtogether/

Using Program Analysis to Improve Database Applications

Alvin Cheung
Samuel Madden Armando Solar-Lezama
MIT CSAIL
{akcheung, madden, asolar}@csail.mit.edu

Owen Arden
Andrew C. Myers
Department of Computer Science
Cornell University
{owen, andru}@cs.cornell.edu

Abstract

Applications that interact with database management systems (DBMSs) are ubiquitous. Such database applications are usually hosted on an application server and perform many small accesses over the network to a DBMS hosted on the database server to retrieve data for processing. For decades, the database and programming systems research communities have worked on optimizing such applications from different perspectives: database researchers have built highly efficient DBMSs, and programming systems researchers have developed specialized compilers and runtime systems for hosting applications. However, there has been relatively little work that optimizes database applications by considering these specialized systems in combination and looking for optimization opportunities that span across them.

In this article, we highlight three projects that optimize database applications by looking at both *the programming system and the DBMS in a holistic manner. By carefully revisiting the interface between the DBMS and the application, and by applying a mix of declarative database optimization and modern program analysis techniques, we show that a speedup of multiple orders of magnitude is possible in real-world applications.*

1 Introduction

From online shopping websites to banking applications, we interact with applications that store persistent data in DBMSs every day. Typically, such applications are written using a general-purpose, imperative language such as Java or Python, with embedded data access logic expressed declaratively in SQL. The application is usually hosted on an application server that is physically separated from (although in close proximity to) the server running the DBMS (we refer to the latter as the database server). During execution, the application issues queries to the DBMS to retrieve or manipulate persistent data.

While this separation between the application and the database helps application development, it often results in applications that lack the desired performance. For example, to achieve good performance, both the compiler and query optimizer optimize parts of the program, but they do not share information, so programmers must manually determine 1) whether a piece of computation should be executed as a query by the DBMS or as general-purpose code in the application server; 2) where computation should take place, as DBMSs are capable of

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
List<User> getRoleUser () {
  List<User> listUsers = new ArrayList<User>();
  List<User> users = ... /* Database query */
  List<Role> roles = ... /* Database query */
  for (int i = 0; i < users.size(); i++) {
    for (int j = 0; j < roles.size(); j++) {
      if (users.get(i).roleId ==
          roles.get(j).roleId) {
        User userok = users.get(i);
        listUsers.add(userok);
  }}}
  return listUsers;
}
```

```
List<User> getRoleUser () {
  List<User> listUsers =
  db.executeQuery(
    "SELECT u
     FROM users u, roles r
     WHERE u.roleId == r.roleId
     ORDER BY u.roleId, r.roleId");
  return listUsers; }
```

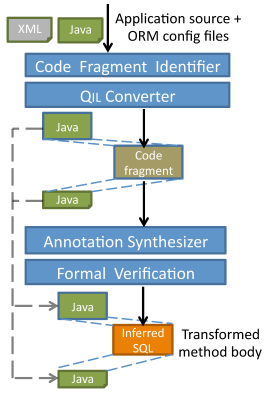


Figure 1: (a) Real-world code example that implements join in Java (left); (b) QBS converted version of code example (middle); (c) QBS architecture (right)

executing general-purpose code using stored procedures, and application servers can similarly execute relational operations on persistent data after fetching them from the DBMS; and 3) how to restructure their code to reduce the number of interactions with the DBMS, as each interaction introduces a network round trip and hence increases application latency.

While handcrafting such optimizations can yield order-of-magnitude performance improvements, these manual optimizations are done at the cost of code complexity and readability. Worse, these optimizations can be brittle. Relatively small changes in control flow or workload pattern can have drastic consequences and cause the optimizations to hurt performance, and slight changes to the data table schema can break manually optimized code. As a result, such optimizations are effectively a black art, as they require a developer to reason about the behavior of the distributed implementation of the program across different platforms and programming paradigms.

Applying recent advances in program analysis and synthesis, we have shown in prior work [10] that we can achieve such optimizations automatically without incurring substantial burden on the developers. In particular, by analyzing the application program, the runtime system, and the DBMS as a whole, along with applying various program analysis and database optimization to drive program optimizations, we have demonstrated order-of-magnitude speedups in real-world database applications. Not only that, we are able to achieve such application improvement while allowing developers to use the same high-level programming model to develop such applications. In this article, we describe three recent projects, in which each project holistically optimizes both the runtime system and the DBMS: QBS, a tool that transforms imperative code fragments into declarative queries to be executed by the DBMS; PYXIS, a system that automatically partitions program logic across multiple servers for optimal performance, and SLOTH, a tool that reduces application latency by eliminating unnecessary network round trips between the application and DBMS servers. In the following we describe each of the projects and highlight representative experimental results.

2 Determining How to Execute

As an example of how database application performance can be substantially improved by analyzing the runtime system and the DBMS together as a whole, consider the code fragment shown in Fig. 1(a). The fragment is adopted from a real-world Java web application that uses the Hibernate Object-Relational Mapping (ORM) library to access persistent data.

As written, the application first issues two queries using Hibernate to fetch a list of user and role objects from the DBMS. The code then iterates the lists in a nested-loop fashion to select the users of interest and returns the list at the end. Note that the code fragment essentially performs a join between two lists. Had

the programming logic been represented as a SQL query, the DBMS query optimizer would be able to take advantage of the indices and join optimizations in choosing the most efficient way to perform the operation. Unfortunately, executing relational operations in the application forgoes all the optimization machinery that DBMSs provide. Programmers frequently write code like this, perhaps because they lack understanding of DBMS functionality, but more often because DBMS operations are abstracted into libraries, such as ORM libraries. While such libraries greatly ease application development, it is difficult for library developers to anticipate how their libraries are used, and application developers to understand the performance implications of their code. As a result, application performance often suffers.

To improve application performance, we would like application frameworks to automatically convert code from one representation to another (e.g., converting from Java to SQL) while presenting the same high-level programming model to the developer as before. Unfortunately, doing so requires a detailed understanding of the database internals. In addition, converting code representations also requires bridging between the application code that is often written in an imperative language such as Java, and database queries that tend to be expressed using a declarative language such as SQL. We are not aware of any application frameworks that perform such cross-system optimizations.

In this section, we describe QBS (Query By Synthesis) [11], a tool that performs such optimization automatically without any user intervention. Given application source code and database configuration, QBS automatically scans the source code to find code fragments that can potentially be converted into SQL to improve performance. For each code fragment, such as the one shown in Fig. 1(a), QBS transforms it into the code shown in Fig. 1(b), where the nested loop is converted into a SQL join query. QBS improves upon prior work on query extraction [29] by recognizing a wide variety of queries in addition to simple selections. In the following, we describe how QBS works and present representative experiment results.

2.1 Converting Program Representations Automatically

QBS uses program synthesis to bridge the gap between imperative and declarative programming paradigms. Fig. 1(c) shows the overall architecture of QBS. Given the source code of the application and database configuration files (the latter is used to determine the classes of objects that are persistently stored and how such objects are mapped to persistent tables), QBS first performs a pre-processing pass to isolate the code fragments that operate on persistent data and contain no other side-effects, such as the one shown in Fig. 1(a). After that, QBS converts each of the identified code fragments into an intermediate code representation called QIL (QBS Intermediate Language). QIL is similar to relational algebra, but is expressive enough to describe operations on ordered lists (which is the data structure that many ORM libraries and JDBC expose for persistent data manipulation). QIL includes operations such as selecting list elements, concatenating two lists, and joining two lists just like in relational algebra, except that the join operates on ordered lists rather than relations.

The resemblance of QIL to relational algebra allows us to easily convert such QIL expressions to SQL queries. However, we still need to reason about the contents of the variables in the code fragment before converting the fragment into SQL. For instance, in the example shown in Fig. 1(a), we need to formally prove that when the nested loop exits, the contents of ”listUsers” is equivalent to that as a result of joining the ”users” and ”roles” lists. QBS uses Hoare-style program reasoning [17] to come up with such proofs. The idea behind coming up with proofs is that if the developer has labeled the contents of each variable in the code fragment (such as ”listUsers”) in terms of other variables, Hoare-style reasoning provides us a way to formally prove that the annotations are indeed correct, and that the expression in Fig. 2(a) indeed represents the value of ”listUsers” at the end of the execution. The problem, of course, is that no such annotations are present in the code. By making use of this observation, however, we can now frame the problem as a search for annotations that can be proven correct using Hoare-style reasoning. Once the annotations are found, translating an expression like the one in Fig. 2(a) into SQL is a purely syntactic process.

A naive strategy for finding the annotations is to search over all possible QIL expressions and check them

$$\mathsf{listUsers} = \pi(\mathsf{sort}(\sigma(\bowtie(\mathsf{users},\mathsf{roles},\mathsf{True}),f_\sigma),l),f_\pi) \quad \begin{array}{rl} & \text{where:} \\ f_\sigma & := \mathsf{get}(\mathsf{users},\mathrm{i}).\mathsf{roleId} = \mathsf{get}(\mathsf{roles},\mathrm{j}).\mathsf{roleId} \\ f_\pi & := \text{projects all the fields from the User class} \\ l & := [\mathsf{users},\mathsf{roles}] \end{array}$$

$$\mathsf{listUsers} = \left\{ \begin{array}{l} \text{expressions involving operators: } \sigma,\pi,\bowtie,\mathsf{sort} \\ \text{and operands: } \mathsf{users},\mathsf{roles} \end{array} \right\}$$

Figure 2: (a) Specification found by QBS (top); (b) potential QIL expressions for `listUsers` from Fig. 1(a) (bottom).

for validity using Hoare-style reasoning. QBS improves upon this strategy using a two-step process. First, QBS analyzes the structure of the input code fragment to come up with a template for the annotations; for example, if the code involves a nested loop, then the annotations are likely to involve a join operator, so this operator is added to the template. By deriving this template from the structure of the code, QBS significantly reduces the space of candidate expressions that needs to be analyzed. In our example, since objects are inserted into ”listUsers” inside a nested loop involving ”users” and ”roles”, the analysis determines that potential QIL expressions for ”listUsers” would involve those two lists as opposed to others. Furthermore, the ”if” statement inside the loop leads the analysis to add selection (but not projection, for instance) as a possible operator involved in the expression, as shown in Fig. 2(b).

After that, QBS performs the search symbolically; instead of trying different expressions one by one, QBS defines a set of equations whose solution will lead to the correct expression. The technology for taking a template and performing a symbolic search over all possible ways to complete the template is the basis for a great deal of recent research in software synthesis. QBS uses this technology through an off-the-shelf system called Sketch [28], which automatically performs the symbolic search and produces the desired annotations. After producing the annotations, QBS uses them to translate part of the code fragment into SQL queries.

2.2 Experimental Results

We implemented a prototype of QBS using the Polyglot compiler framework [21]. Our prototype takes in Java source code that uses the Hibernate ORM library to interact with DBMSs, identifies code fragments that can be converted into SQL expressions, and attempts to convert them into SQL using the algorithm described above.

We tested the ability of QBS to transform real-world code fragments. We used QBS to compile 120k lines of open-source code written in two applications: a project management application Wilos [4] with 62k LOC, and a bug tracking system itracker [1] with 61k LOC. Both applications are written in Java using Hibernate. QBS first identified the classes that are persistently stored, and scanned through the application for code fragments that use such classes, such as that in Fig. 1(a). This resulted in 49 benchmark code fragments in total. QBS then attempted to rewrite parts of each benchmark into SQL. Fig. 3 shows the number of fragments that QBS was able to convert into relational equivalents. We broadly categorize the code fragments according to the type of relational operation that is performed. While many fragments involve multiple relational operations (such as a join followed by a projection), we only give one label to each fragment in order not to double count.

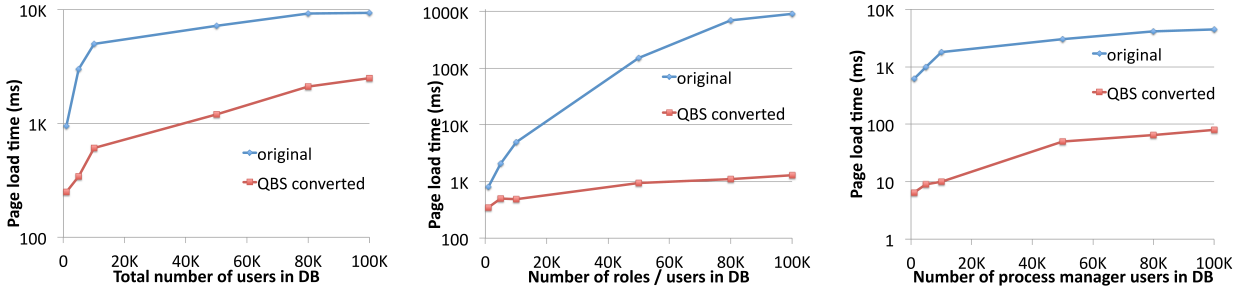
The result shows that QBS is able to recognize and transform a variety of relational operations, including selections, joins, and aggregations such as finding max and min values, sorting, and counting. The slowest transformation of any individual benchmark took 5 minutes to complete, such as the one shown in Fig. 1(a) as it involves join operations, and the average was around 3 minutes. The largest benchmarks involve around 200 lines of source code, and the majority of the time was spent in the annotation search process.

Our QBS prototype currently only handles read-only persistent operations, and as a result cannot handle code that updates to inserts persistent data. In addition, there were a few benchmarks involving read-only operations

Wilos (project management application – 62k LOC)		
operation type	# benchmarks	# translated by QBS
projection	2	2
selection	13	10
join	7	7
aggregation	11	10
total	**33**	**29**

itracker (bug tracking system – 61k LOC)		
operation type	# benchmarks	# translated by QBS
projection	3	2
selection	3	2
join	1	1
aggregation	9	7
total	**16**	**12**

Figure 3: QBS experiment results from real-world benchmarks conversions



(a) Selection with 50% selectivity (b) Join code fragment from Fig. 1(a) (c) Aggregation code fragment

Figure 4: Webpage load times comparison of representative code fragments

that QBS were not able to transform. Some of these benchmarks use type information in program logic, such as storing polymorphic records in the database and performing different operations based on the type of records retrieved. Including type information in ordered lists should allow QBS to process most of the benchmarks. Meanwhile, some benchmarks re-implement list operations that are provided by the JDK (such as sorting or computing the max value) in a custom manner. Extending QIL expressions to include such operations will allow QBS to convert such benchmarks.

We also compared the load times for web pages containing the benchmarks before and after QBS conversion. The results from representative code fragments are shown in Fig. 4, with Fig. 4(b) showing the results from the code example from Fig. 1(a). In all cases, performance improved in part because the application only needs to fetch the query results from the DBMS. Moreover, as in the case of the code example in Fig. 1(a), expressing the program logic as a SQL query allows the DBMS to execute the join using efficient join algorithms, as opposed to executing the join in a nested loop fashion as in the original code, which leads to further speedup. The aggregate code fragments illustrate similar behavior.

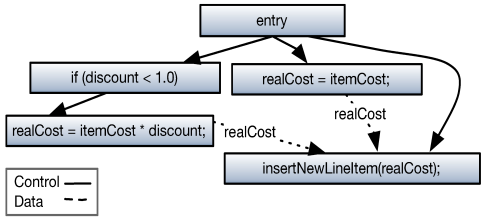
3 Improving Locality With Program Partitioning

In addition to deciding how application logic should be expressed, determining *where* it should be executed is another important factor that affects application performance. *Automatic program partitioning* is a technique for splitting a program into communicating processes that implement the original program in a semantics-preserving way. While prior applications of automatic program partitioning include optimizing distributed applications [18, 20, 31, 13, 14, 5], parallelizing sequential programs [25, 19], and implementing security policies and principles [32, 12, 6, 30], we are not aware of prior work that applies program partitioning to improve database application performance. Partitioning database applications is challenging as the optimal solution often depends on the current server workload. To that end, we developed PYXIS [8], a tool that automatically partitions a database application between the application and database servers, and adaptively changes how the application is split based on the amount of resources available on the database server.

PYXIS consists of two components: a compiler and a runtime Java library. The PYXIS compiler assigns the statements and fields in Java classes to either an application server or a database server. We call the code

```
realCost = itemCost;
if (discount < 1.0)
    realCost = itemCost * discount;
insertNewLineItem(realCost);
```

(a) An example code fragment



(b) Dependency graph

Figure 5: Illustration of the data structures used in program partitioning

and data assigned to a particular host a *partition* and the set of partitions a *partitioning*. Placing code that accesses the DBMS on the database server results in lower latency since it requires fewer network round trips. However, running more code on the database server places additional computational load on the server, which could negatively impact latency and reduce throughput. PYXIS manages this tradeoff by finding partitionings that minimize overall workload latency within a specified budget. This budget limits the impact on server load.

Since the optimal partitioning depends on the current workload, the PYXIS runtime adapts to changing workload characteristics by switching between partitionings dynamically. Given a set of partitionings with varying budgets, PYXIS alleviates load on the database server by switching to a partitioning with a lower budget that places more code at the application server.

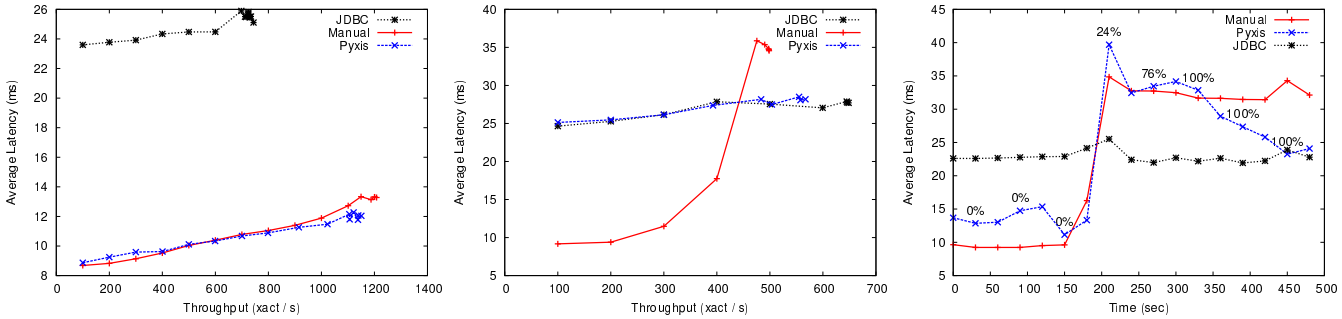
3.1 Analyzing Dependencies in Database Applications

At the core of the PYXIS compiler is a program partitioner. Program partitioning is done using a static interprocedural dependency analysis based on an object-sensitive pointer analysis [27]. The analysis constructs a dependency graph in which each statement of the program is a node and each edge represents a possible dependency between statements. PYXIS distinguishes two kinds of dependencies: control and data. Control dependency edges connect statements to branch points in control flow such as ”if” statements, method calls, or exception sites. Data dependencies connect uses of heap data with their possible definitions. Fig. 5a shows some example code and Fig. 5b its corresponding dependency graph. The statement at line 3 is *control dependent* on the ”if” condition at line 2. The statement at line 4 is *data-dependent* on the two statements that make assignments to ”realCost” at line 1 and 3.

The PYXIS compiler transforms the input program into two separate programs that share a distributed heap. A novel aspect of PYXIS is that the runtime is designed to allow the partitioner to take advantage of the static analysis results to generate optimized code. To this end, all synchronization of the distributed heap is performed explicitly by custom code generated for each application. When executing a block of instructions, all updates to the distributed heap are batched until control is transferred to the remote node. At this point, updates that may be observed by the remote runtime are transferred and execution continues at the remote node. By only transferring the observed updates, PYXIS significantly reduces communication overhead for heap synchronization. The static analysis conducted by the partitioner is conservative, ensuring that all heap locations accessed are guaranteed to be up to date, even though other parts of the heap may contain stale data.

3.2 Partitioning the input program

We now discuss the mechanics involved in partitioning the input program. First, PYXIS gathers profile data such as code coverage and data sizes from an instrumented version of the application under a representative workload. This data is used to weight the dependency graph. We call this weighted dependency graph a *partition graph*. Each statement node in the graph is annotated with how many times it was executed in the workload. Each edge in the graph is annotated with an estimated cost for transferring control or data between the application and



(a) High-budget latency vs throughput (b) Low-budget latency vs throughput (c) Latency over time with switching

Figure 6: TPC-C experiment results on a 16-core database server

database server should the two associated statements be placed separately.

The weights in the partition graph encode the cost model of the PYXIS runtime. Since updates are batched, the weights of data dependencies are relatively cheap compared to control dependencies. Node weights capture the relative load incurred server for different statements—more frequently executed statements typically result in higher associated load.

Next, we determine how to optimally cut the partition graph. A *cut* in the partition graph is an assignment of statements to either the database or the application server. The best cut should minimize the weights of the cut edges while obeying certain constraints. Among these constraints are the budget constraint: the sum of node weights assigned to the database server cannot exceed the amount of resources available on the server. The problem of finding the best cut is translated to a binary integer programming problem and sent to a solver. Once the solver finds a solution, the partitioner outputs the two components that make up the partitioned program. One component runs at the application server and one at the database. Note that all partitionings are generated offline by setting different values for the resource available on the database server.

The partitioned graph is then compiled to code. Statements are grouped into execution blocks that share the same placement and control dependencies. The exit points in each execution block specify a local or remote block where the computation proceeds. For instance, say the call to ”insertNewLineItem” in Fig. 5a is placed on the database server while the other statements remain at the application server. Then, when control is transferred from the application server to the database, the current value of ”realCost” will be transferred as well, ensuring that the call’s parameter is up to date.

3.3 Experimental Results

We built a prototype of PYXIS using Polyglot [21] and the Accrue analysis framework [3]. We have evaluated our implementation on TPC-C and TPC-W. We used PYXIS to create two partitionings—one using a low CPU budget and one using a high CPU budget. We compared the PYXIS generated programs to a traditional client-side implementation that executes all application logic on the application server and uses JDBC to execute queries (referred to as **JDBC**), as well as a version that encodes most application logic in stored procedures that are executed on the database server (referred to as **Manual**).

Our experiments show that PYXIS automatically generated partitionings that were competitive with Manual in performance. In Fig. 6a, a high-budget partition was evaluated in which the database server has extra CPU resources. In this case, the PYXIS-generated program performed similarly to Manual. This is because much of the program code—especially methods with multiple SQL queries—could be placed on the database server, and latency was low. In Fig. 6b, a low-budget partition was evaluated in a context where the database server has limited CPU resources. Here, the PYXIS-generated program performed like JDBC, where all application statements other than database queries are placed at the application server. Latency was higher than Manual at

low throughput, but because less computational load was placed on the database, the PYXIS-generated program was able to achieve a higher throughput.

In general, the availability of CPU resources may change as the workload evolves. Consequently, a particular partitioning may not perform adequately across all workload profiles. Figure 6c shows the results of an experiment that evaluates the ability of PYXIS to adapt to changing resource availability. In this experiment, the load on the database server started out small, but was artificially increased after three minutes. The Manual implementation performed well initially, but its latency exceeded that of the JDBC implementation as the amount of CPU resources on the database server decreased. PYXIS, on the other hand, first chose to serve the incoming requests using a partitioning similar to Manual. But as CPU resources decrease, the PYXIS runtime monitors the load at the server and switches to a partitioning similar to JDBC instead when the load exceeds a threshold. Thus, PYXIS obtains the minimal latencies of both implementations at all times.

A single application may be subjected to several different performance profiles. By examining the DBMS and application server together, we are able to generate program partitionings specialized for each profile using PYXIS. In sum, PYXIS enhances the performance and scalability of database applications without forcing developers to make workload-specific decisions about where code should be executed or data should be stored. It would be interesting to use PYXIS to partition larger programs, and the same technique to other application domains beyond database applications.

4 Batching DBMS Interactions

PYXIS shows that reducing round trips between the application and database servers can improve application performance significantly. *Query batching* is another common technique to reduce such round trips. For instance, many ORM libraries allow developers annotate their code to prefetch extra persistent objects before such objects are needed by the program. Unfortunately, deciding when and what objects to prefetch is difficult, especially when developers do not know how the results that are returned by their methods will be used.

There is prior research that uses static program analysis to extract queries that will be executed unconditionally by a single client in the application [16, 23, 22]. The extracted queries are executed asynchronously in a single round trip when all query parameters are computed [7]. There has also been work on multi-query optimization [15, 26, 24] that focuses on sharing query plans or reordering transactions among multiple queries. However, such approaches aim to combine queries issued by multiple concurrent clients at the database rather than in the application.

Rather than prefetching of query results, where its effect is often limited by the imprecision in static analysis, instead we aim to *delay* queries as long as possible, with the goal to batch the delayed queries so that they can be issued in a single round trip to the DBMS. In this section we describe SLOTH [9], a new system we built to achieve this goal. Instead of using static analysis, SLOTH creates query batches by using *extended lazy evaluation*. As the application executes, queries are batched into a *query store* instead of being executed right away. In addition, non-database related computation is delayed until it is absolutely necessary. As the application continues to execute, multiple queries are accumulated with the query store. When a value that is derived from query results is finally needed (say, when it is printed on the console), then all the queries that are registered with the query store are executed by the database in a single batch, and the results are then used to evaluate the outcome of the computation. In the following we describe SLOTH in detail and highlight some experiment results using real-world database applications.

4.1 Extending Lazy Evaluation for Query Batching

SLOTH builds upon traditional lazy evaluation for query batching. Lazy evaluation, as pioneered by functional languages, aims to delay computation until its results are needed. As mentioned, in SLOTH we extend lazy

```
1  ModelAndView handleRequest(...) {
2  Map model = new HashMap<String, Object>();
3  Object o = request.getAttribute("patientId");
4  if (o != null) {
5    Integer patientId = (Integer) o;
6    if (!model.containsKey("patient")) {
7      if (hasPrivilege(VIEW_PATIENTS)) {
8        Patient p = getPatientService().getPatient(patientId);
9        model.put("patient", p);
10       ...
11       model.put("patientEncounters",
12         getEncounterService().getEncountersByPatient(p));
13       ...
14       List visits = getVisitService().getVisitsByPatient(p);
15       CollectionUtils.filter(visits, ...);
16       model.put("patientVisits", visits);
17       model.put("activeVisits", getVisitService().
18         getActiveVisitsByPatient(p));
19   ...
20   return new ModelAndView(portletPath, "model", model);
21 }
```

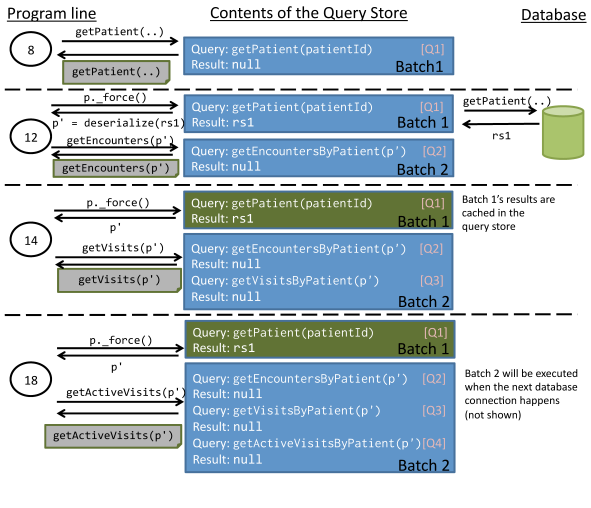


Figure 7: (a) Code fragment abridged from OpenMRS (left); (b) How SLOTH executes the code fragment (right)

evaluation where computation is deferred, except that when queries are encountered they are batched in a query store. When any of the query results are needed, all batched queries are executed in a single interaction with the DBMS. Using extended lazy evaluation allows us to batch queries across conditional statements and even method boundaries without any extra work for the developer, as queries that are accumulated in the query store are guaranteed to be executed by the application. Moreover, the batched queries are *precise* in that they are those that were issued by the application.

To understand extended lazy evaluation in action, consider the code fragment in Fig. 7(a), which is abridged from a real-world Java web application [2] that uses Spring as the web framework and the Hibernate ORM library to manage persistent data.

The application is structured using the Model-View-Control design pattern, and the code fragment is part of a controller that builds a model to be displayed by the view after construction. The controller is invoked by the web framework when a user logs-in to the application to view the dashboard for a particular patient. The controller first creates a model (a ”HashMap” object), populates it with appropriate patient data based on the logged-in user's privileges, and returns the populated model to the web framework. The web framework then passes the partially constructed model to other controllers which may add additional data, and finally to the view creator to generate HTML output (code not shown).

As written, the code fragment can issue up to four queries; the queries are issued by calls of the form ”getXXX” following the web framework's convention. The first query in Line 8 fetches the ”Patient” object that the user is interested in displaying and adds it to the model. The code then issues queries on Lines 12 and 14, and Line 18 to fetch various data associated with the patient, and adds the data to the model as well. It is important to observe that of the four round trips that this code can incur, only the first one is essential—without the result of that first query, the other queries cannot be constructed. In fact, the results from the other queries are only stored in the model and not used until the view is actually rendered. Thus, the developer could have collected in a single batch all the queries involved in building the model until the data from any of the queries in the batch is really needed—either because the model needs to be displayed, or because the data is needed to construct a new query. Manually transforming the code in this way would have a big impact in the number of round trips incurred by the application, but unfortunately would also impose an unacceptable burden on the developer.

Using SLOTH, the code fragment is compiled to execute lazily, where the evaluation of a statement does not cause it to execute; instead, the evaluation produces a *Thunk*: a place-holder that stands for the result of that computation, and it also remembers what the computation was. As mentioned, when executing statements

that issue queries, the queries themselves are batched in the query store in addition to a thunk being returned. Meanwhile, the only statements that are executed immediately upon evaluation are those that produce output (e.g., printing on the console), or cause an externally visible side effect (e.g., reading from files or committing a DBMS transaction). When such a statement executes, the thunks corresponding to all the values that flow into that statement will be *forced*, meaning that the delayed computation they represented will finally be executed.

This is illustrated in Fig. 7(b). Line 8 issues a call to fetch the "Patient" object that corresponds to "patientId" (Q1). Rather than executing the query, SLOTH compiles the call to register the query with the query store instead. The query is recorded in the current batch within the store (Batch 1), and a thunk is returned to the program (represented by the gray box in the figure). Then, in Line 12, the program needs to access the patient object "p" to generate the queries to fetch the patient's encounters (Q2) followed by visits in Line 14 (Q3). At this point the thunk "p" is forced, Batch 1 is executed, and its results ("rs1") are recorded in the query cache in the store. A new non-thunk object "p'" is returned to the program upon deserialization from "rs1", and "p'" is memoized in order to avoid redundant deserializations. After this query is executed, Q2 and Q3 can be generated using "p'" and are registered with the query store in a new batch (Batch 2). Unlike the patient query, however, Q2 and Q3 are not executed within the code fragment since their results are not used (thunks are stored in the model map in Lines 12 and 16). Note that even though Line 15 uses the results of Q3 by filtering it, SLOTH determines that the operation does not have externally visible side effects and is thus delayed, allowing Batch 2 to remain unexecuted. This leads to batching another query in Line 18 that fetches the patient's active visits (Q4), and the method returns.

Depending on subsequent program path, Batch 2 might be appended with further queries. Q2, Q3, and Q4 may be executed later when the application needs to access the database to get the value from a registered query, or they might not be executed at all if the application has no further need to access the database. Using SLOTH, the number of DBMS round trips is reduced from four to one during execution of the code fragment.

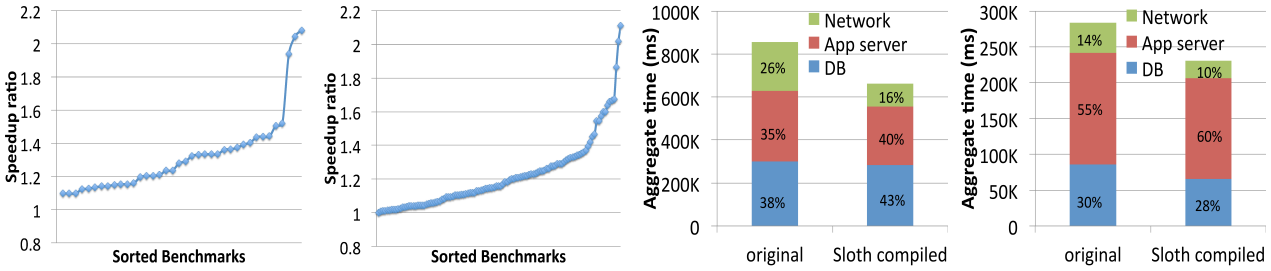
4.2 Evaluating SLOTH

We have built a prototype of SLOTH. The protoype takes in Java source code and compiles it to be executed using extended lazy evaluation. SLOTH also comes with a runtime library that includes the implementation of the query store, along with a custom JDBC driver that allows multiple queries to be issued to the database in a single round trip, and extended versions of the web application framework, ORM library, and application server to process thunks (we currently provided extensions to the Spring application framework, the Hibernate ORM library, and the Tomcat application server.) Note that among the changes to existing infrastructure, only the custom JDBC driver and extended ORM library are essential. The other extensions are included in order to increase batching opportunities.

We evaluated our prototype using two real-world applications: itracker (the same as used in evaluating QBS), and OpenMRS, with a total of 226k lines of Java code. Both applications already make extensive use of prefetching annotations provided by Hibernate. We created benchmarks from the two applications by manually examining the source code to locate all web page files (html and jsp files). Next, we analyzed the application to find the URLs that load each of the web pages. This resulted in 38 benchmarks for itracker, and 112 benchmarks for OpenMRS. Each benchmark was run by loading the extracted URL from the application server via a client that resides on the same machine as the server. There is a 0.5ms round trip delay between the two machines.

For the experiments, we compiled the two applications using SLOTH, and measured the end-to-end page load times for the original and the SLOTH-compiled benchmarks. Fig. 8(a) and (b) show the load time ratios between the SLOTH-compiled and original benchmarks.

The results show that the SLOTH-compiled applications loaded the benchmarks faster compared to the original applications, achieving up to 2.08× (median 1.27×) faster load times for itracker and 2.1× (median 1.15×) faster load times for OpenMRS. There were cases where the total number of queries decreased in the SLOTH-compiled version as they were issued during model creation but were not needed by the view. Most of the



(a) itracker benchmarks load time ratios (b) OpenMRS benchmarks load time ratios (c) itracker benchmarks timing breakdown (d) OpenMRS benchmarks timing breakdown

Figure 8: SLOTH experiment results

speedups, however, resulted from batching multiple queries and reducing round trips to the DBMS (up to 68 queries were batched in one of the benchmarks). To quantify the latter, we tabulated the aggregate amount of time spent in each processing step across all the benchmarks. The results are shown in Fig. 8(c) and (d), which show that the time spent in network traffic was significantly decreased in the SLOTH-compiled benchmarks.

Overall, the experiments show SLOTH improves application performance significantly despite considerable developer efforts spent in annotating objects for prefetching. It would be interesting to combine SLOTH with other prefetching techniques. For instance, SLOTH currently pauses the application in order to execute the batched queries. Alternatively, such batched queries can be executed asynchronously [7] once they are identified in order to avoid pausing the application. It would also be interesting to investigate query rewrite techniques (such as query chaining [23]) for the batched queries rather than issuing them in parallel at the database as in the current SLOTH prototype.

5 Conclusion

In this article, we described three projects that improve the performance of database applications. We showed that by co-optimizing the runtime system and the DBMS, we can achieve a speedup of multiple orders of magnitude in application performance. We believe that the techniques presented are complementary to each other. In future work, we plan to investigate combining these techniques and measuring the overall performance improvement using real-world applications.

The idea of co-optimization across different software systems opens up new opportunities in both programming systems and database research. For instance, it would be interesting to apply program partitioning to other application domains that involve multiple systems, such as partitioning code and data among web clients, application servers, and the DBMS. In addition, techniques used in QBS can also be used to build new optimizing compilers: for instance, converting code to make use of specialized routines such as MapReduce or machine learning libraries.

References

[1] itracker Issue Management System. `http://itracker.sourceforge.net/index.html`.

[2] OpenMRS medical record system. `http://www.openmrs.org`.

[3] The Accrue Analysis Framework. `http://people.seas.harvard.edu/~chong/accrue.html`.

[4] Wilos Orchestration Software. `http://www.ohloh.net/p/6390`.

[5] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 273–286, 2003.

[6] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. USENIX Security Symposium*, pages 57–72, 2004.

[7] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 375–386, 2011.

[8] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *PVLDB*, 5(11):1471–1482, 2012.

[9] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2014.

[10] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *Conference on Innovative Data Systems Research (CIDR)*, Jan. 2013.

[11] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 3–14, 2013.

[12] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. *Comm. of the ACM*, (2), 2009.

[13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proc. EuroSys*, pages 301–314, 2011.

[14] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 49–62, 2010.

[15] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[16] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *PVLDB*, 1(1):1107–1123, 2008.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, 1969.

[18] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 187–200, 1999.

[19] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 170–187, 2004.

[20] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 395–408, 2009.

[21] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *Proc. International Conference on Compiler Construction (CC)*, pages 138–152, 2003.

[22] K. Ramachandra and R. Guravannavar. Database-aware optimizations via static analysis. *IEEE Data Engineering Bulletin*, 37(1), Mar. 2014.

[23] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 133–144, 2012.

[24] S. Roy, L. Kot, and C. Koch. Quantum databases. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[25] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5.6):779–804, Sept. 1991.

[26] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

[28] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 167–178, 2007.

[29] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 19–36, 2008.

[30] Y. Wu, J. Sun, Y. Liu, and J. S. Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 323–333, 2013.

[31] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 32–43, 2006.

[32] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, May 2003.

Database-Aware Program Optimizations via Static Analysis

Karthik Ramachandra
I.I.T. Bombay
karthiksr@cse.iitb.ac.in

Ravindra Guravannavar
Independent Consultant
ravig@acm.org

Abstract

Recent years have seen growing interest in bringing together independently developed techniques in the areas of optimizing compilers and relational query optimization, to improve performance of database applications. These approaches cut across the boundaries of general purpose programming languages and SQL, thereby exploiting many optimization opportunities that lie hidden both from the database query optimizer and the programming language compiler working in isolation. Such optimizations can yield significant performance benefits for many applications involving database access. In this article, we present a set of related optimization techniques that rely on static analysis of programs containing database calls, and highlight some of the key challenges and opportunities in this area.

1 Introduction

Most database applications are written using a mix of imperative language constructs and SQL. For example, database applications written in Java can execute SQL queries through interfaces such as JDBC or Hibernate. Database stored procedures, written using languages such as PL/SQL and T-SQL, contain SQL queries embedded in imperative program code. Such procedures, in addition to SQL, make use of assignment statements, conditional control transfer (IF-ELSE), looping and calls to subroutines. Further, SQL queries can make calls to user-defined functions (UDFs). User-defined functions can in turn make use of both imperative language constructs and SQL.

Typically the imperative program logic is executed outside the database query processor. Queries embedded in the program are submitted (typically synchronously, and over the network) at runtime, to the query processor. The query processor explores the space of alternative plans for a given query, chooses the plan with the least estimated cost and executes it. The query result is then sent back to the application layer for further processing. A database application would typically have many such interactions with the database while processing a single user request. Such interactions between the application and the database can lead to many performance issues that go unnoticed during development time. Traditional optimization techniques are either database centric (such as query optimization and caching), or application centric (such as optimizations performed by compilers), and do not optimize the interactions between the application and the database. This is because neither the programming language compiler nor the database query processor gets a global view of the application. The language compiler treats calls to the database as black-box function calls. It cannot explore alternative plans for executing a single query or a set of queries. Similarly, the database system has no knowledge of the context in

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Example 1 An opportunity for database aware optimizations

```
int sum = 0;
stmt = con.prepareStatement("select count(partkey) from part where p_category=?");
while(!categoryList.isEmpty()) {
    category = categoryList.removeFirst();
    stmt.setInt(1, category);
    ResultSet rs = stmt.executeQuery();
    int count = rs.getInt("count");
    sum += count;
}
```

which a query is being executed and how the results are processed by the application logic, and has to execute queries as submitted by the application.

For instance, in database applications, loops with query execution statements such as the Java program snippet in Example 1 are quite commonly encountered. Such loops result in repeated and synchronous execution of queries, which is a common cause for performance issues in database applications. This leads to a lot of latency at the application due to the many network round trips. At the database end, this results in a lot of random IO and redundant computations. A program compiler or a database query execution engine working independent of each other cannot reduce the latency or random IO in Example 1. Usually such loops are manually optimized by rewriting them to use set oriented query execution or asynchronous prefetching of results. Set oriented execution reduces random disk IO at the database, and also reduces network round trips. The effect of network and IO latency can be reduced by overlapping requests using asynchronous prefetching.

However, manually performing such transformations is tedious and error prone. It is very difficult to identify such opportunities in complex programs. These transformations can be automated by performing a combined analysis of the program along with the queries that it executes. In this paper, we give an overview of the techniques that we have developed as part of the DBridge project at IIT Bombay. This is joint work with Mahendra Chavan and S Sudarshan. Our techniques automatically optimize applications by rewriting both the application code and queries executed by the application together, while preserving equivalence with the original program. Such transformations can either be implemented as part of a *database-aware* optimizing compiler or as a plug-in for any integrated development environment (IDE) providing a visual, source-to-source application rewrite feature.

The paper is organized as follows. In Sections 2, 3 and 4, we describe various program transformation techniques, and show how they work together in order to optimize the program in Example 1. In Section 5, we briefly describe the design of DBridge, a research prototype in which we have implemented the techniques presented. Related work is briefly discussed in Section 6. In Section 7, we describe challenges and open problems in this area, and conclude in Section 8.

2 Set-Oriented Query Execution

Loops containing query execution statements, such as the one in Example 1 are often found to be performance bottlenecks. Such loops suffer poor performance due to the following reasons: (a) they perform multiple round trips to the database, (b) the database system is required to process one query at a time, as a result of which, it cannot use efficient set-oriented algorithms (such as hash or merge joins), which would compute answers for all the queries in one go and (c) per query overheads, such as parameter validation and authorization checks, are incurred multiple times. Due to these reasons, loops containing query execution statements are often the most promising targets for program transformations to improve performance.

Example 2 Transformed program for Example 1 after loop fission [11]

```
int sum = 0;
PreparedStatement stmt = con.prepare("select count(partkey) from part where category=?");
LoopContextTable lct = new LoopContextTable();
while(!categoryList.isEmpty()) {
    LoopContext ctx = lct.createContext();
    category = categoryList.removeFirst();
    ctx.setInt("category", category);
    stmt.setInt(1, category);
    stmt.addBatch(ctx);
}
stmt.executeBatch();
for(LoopContext ctx : lct) {
    category = ctx.getInt("category");
    ResultSet rs = stmt.getResultSet(ctx);
    int count = rs.getInt("count");
    sum += count;
}
```

Repeated execution of a query by a loop can often be avoided by rewriting the query into its *set-oriented form* and moving it outside the loop. Given a parameterized query $q(r)$, its set-oriented form $q_b(rs)$ is a query whose result contains the result of $q(r)$ for every r in the parameter set rs [11]. Using set-oriented forms of queries avoids loss of performance due to afore-mentioned reasons. However, automating such a loop transformation requires analyzing the data dependences between statements in the loop body. Query parameters are often values produced by other statements in the loop, and query results are used by other statements in the loop. Such statements must be suitably rewritten. Loop transformation for set-oriented query execution is described in [11, 10], and consists of two key steps: (i) loop distribution (loop fission) to form a canonical query execution loop, and (ii) replacing the canonical query execution loop with the set-oriented form of the query.

Loop distribution splits a loop into multiple loops, each containing a subset of the statements in the original loop. It is applied so that query execution statements inside the loop are isolated into separate canonical query execution loops. A canonical query execution loop is a loop that can be entirely replaced by a single query execution statement. In [11] a canonical query execution loop is a loop that (a) executes a query $q(r)$ for each record r in set rs, (b) contains no statements other than the query execution statement, and (c) results of the query $q(r)$ are stored along with the record r in rs. A canonical query execution loop containing query $q(r)$ is then replaced with a statement that executes the set-oriented query $q_b(rs)$.

The final rewritten program for Example 1 after performing loop fission is shown in Example 2. The first loop in the transformed program builds a temporary table with all the parameter bindings (using the *addBatch* API). Next, a rewritten form of the query is executed (using the *executeBatch* API) to obtain results for all the parameter bindings together. Then, the second loop executes statements that depend on the query results. The scalar aggregate query in Example 1 would be transformed into the following query, where *pb* is a temporary table in which the parameter bindings are materialized.

```
SELECT pb.category, le.c1 FROM pbatch pb,
    OUTER APPLY (SELECT count(partkey) as c1
            FROM part WHERE category=pb.category) le;
```

The rewritten query uses the OUTER APPLY construct of Microsoft SQL Server but can also be written using a left outer join combined with the LATERAL construct of SQL:99. Most widely used database systems

can decorrelate such a query into a form that uses joins or outer joins [9].

Queries, being side-effect free, can be executed in any order of the parameter bindings. However, the loop can contain other order-sensitive operations, which must be executed in the same order as in the original program. The *LoopContextTable* data structure ensures the following: (i) it preserves the order of execution between the two loops and (ii) for each iteration of the first loop, it captures the values of all variables updated, and restores those values in the corresponding iteration of the second loop. The rewritten program not only avoids the overheads of multiple round-trips, but also enables the database system to employ an efficient algorithm (such as hash/sort based grouping) to evaluate the result more efficiently. This loop transformation for set-oriented execution can be applied even when a query is conditionally executed inside an *if-then-else* statement. Queries inside multiple levels of nested loops can also be pulled out and replaced with their set-oriented forms. Details of the transformations can be found in [11, 10].

Statement Reordering: Data dependencies [15] between statements inside the loop play a crucial role in the loop distribution transformation. Loop distribution cannot be directly applied when there exist *loop-carried* flow dependencies (also known as write-read or true dependencies) between statements across the loop split boundaries. However, in many cases, by introducing additional variables and reordering the statements, it is possible to eliminate loop-carried dependencies that hinder the transformation. The algorithm for reordering statements so as to eliminate loop-carried flow dependencies crossing the loop split boundaries can be found in [10]. The desired reordering is possible if the query execution statement of interest does not belong to a cycle of true data dependencies. After reordering the statements, the loop fission transformation can be applied.

3 Asynchronous and Batched Asynchronous Query Submission

As described in Section 2, batching can provide significant benefits because it reduces the delay due to multiple round trips to the database and allows more efficient query processing techniques to be used at the database. Although batching is very beneficial, it does not overlap client computation with that of the server, as the client blocks after submitting the batch. Batching also results in a delayed response time, since the initial results from a loop appear only after the complete execution of the batch. Also, batching may not be applicable altogether when there is no efficient set-oriented interface for the request invoked, as is the case for many Web services.

As compared to batching, asynchronous prefetching of queries can allow overlap of client computation with computation at the server; it can also allow initial results to be processed early, instead of waiting for an entire batch to be processed at the database, which can lead to better response times for initial results. Asynchronous submission is also applicable if the query executed varies in each iteration of a loop. In this section, we focus on performing asynchronous prefetching in loops; Section 4 additionally describes a technique to handle procedure calls and straight line code. Opportunities for asynchronous submission are often not very explicit in code. For the program given in Example 1, the result of the query, assigned to the variable *count*, is needed by the statement that immediately follows the assignment. For the code in its present form there would be no gain in replacing the blocking query execution call by a non-blocking call, as the execution will have to block immediately after making a non blocking submission.

However, it is possible to automatically transform the given loop to enable beneficial asynchronous query submission. The transformations described in the context of batching in Section 2 can be extended to exploit asynchronous submission, as presented in [5]. As shown in Example 2, the loop in Example 1 is split at the point of query execution. However, instead of building a parameter batch and executing a set-oriented query, the program in Example 2 can instead perform asynchronous query submissions as follows.

The first loop of Example 2 invokes the *addBatch* method in each iteration. In the asynchronous mode, this *addBatch* method is modeled as a non blocking function that submits a request onto a queue and returns immediately (this method could be called *submitQuery* instead, but we stick to *addBatch* so that the decision to perform batching or asynchronous submission can be deferred till runtime). This queue is monitored by a thread

pool, and requests are picked up by free threads which execute the query in a synchronous manner. The results are then placed in a cache keyed by the loop context(ctx). The *executeBatch()* invocation does nothing in the case of asynchronous submission and can be omitted. In the second loop of Example 2, the program invokes *getResultSet*, which is a blocking function. It first checks the cache if the results are already available, and if not, blocks till they become available. More details regarding this transformation can be found in [5], and the design of the asynchronous API is discussed in Section 5. There are further extensions and optimizations to this technique, and we now discuss one such extension briefly.

Asynchronous Batching: Although asynchronous submission can lead to significant performance gains, it can result in higher network overheads, and extra cost at the database, as compared to batching. Batching and Asynchronous submission can be seen as two ends of a spectrum. Batching, at one end, combines all requests in a loop into one big request with no overlapping execution, where as asynchronous submission retains individual requests as is, while completely overlapping their execution. There is a range of possibilities between these two, that can be achieved by *asynchronous* submission of multiple, smaller *batches* of queries. This approach, called *asynchronous batching*, retains the advantages of batching and asynchronous submission, while avoiding their drawbacks. This is described in detail in [16], and we summarize the key ideas here.

As done in pure asynchronous submission, the non-blocking *addBatch* function places requests onto a queue. In pure asynchronous submission however, each free thread picks up one pending request from the queue. Instead, we now allow a free thread to pick up multiple requests, which are then sent as a batch, by rewriting the queries as done in batching. The size of these batches, and the number of threads to use, are parameters that can be adaptively tuned at runtime, based on metrics such as the arrival rate of requests onto the queue, and the request processing rate [16].

Further, observe that in Example 2, the processing of query results (the second loop) starts only after all asynchronous submissions are completed i.e, after the first loop completes. Although this transformation significantly reduces the total execution time, it results in a situation where results start appearing much later than in the original program. In other words, for a loop of n iterations, the time to k-th response ($1 \leq k \leq n$) for small k is more as compared to the original program, even though the time may be less for larger k. This could be a limitation for applications that need to show some results early, or that only fetch the first few results and discard the rest. This limitation can be overcome by overlapping the consumption of query results with the submission of requests. The transformation can be extended to run the producer loop (the loop that makes asynchronous submissions) as a separate thread. That is, the main program spawns a thread to execute the producer loop, and continues onto the second loop immediately. The details of this extension are given in [16].

Asynchronous batching can achieve the best of batching and asynchronous submission, since it has the following characteristics.

- Like batching, it reduces network round trips, since multiple requests may be batched together.
- Like asynchronous submission, it overlaps client computation with that of the server, since batches are submitted asynchronously.
- Like batching, it reduces random IO at the database, due to use of set oriented plans.
- Although the total execution time of this approach might be comparable to that of batching, this approach results in a much better response time comparable to asynchronous submission, since the results of queries become available much earlier than in batching.
- Memory requirements do not grow as much as with pure batching, since we deal with smaller batches.

4 Prefetching of query results

Consider a loop that invokes a procedure in every iteration, such as the one in Example 3. The loop invokes the procedure *computePartCount*, which in turn executes a query. Both asynchronous query submission and

Example 3 Opportunity for prefetching across procedure invocations

```
rs = executeQuery("select category_id from categories where cat_group=?", group_id);
while(rs.next()) {
    int category = rs.getInt("category_id");
    partCount = computePartCount(category, flag);
    sum += partCount;
}
int computePartCount(int category, boolean flag) {
    int count = 0;
    limit = DEFAULT;
    if(flag) limit = DEFAULT * 2;
    // some computations
    rs2 = executeQuery("select count(partkey) as part_count from part where p_category=?", category);
    if (rs2.next() {
       count = rs2.getString("part_count");
       if (count > limit) count = limit;
    }
    return count;
}
```

Example 4 The program of Example 3 after inserting a prefetch submission

```
rs = executeQuery("select category_id from categories where cat_group=?", group_id);
while(rs.next()) {
    category = rs.getInt("category_id");
    submitQuery("select count(partkey) as part_count from part where p_category=?", category);
    partCount = computePartCount(category); // this function remains unchanged as in Example 3 except that
                        // the executeQuery() first looks up a cache, and blocks if results are not yet available
    sum += partCount;
}
```

batching depend on the loop fission transformation which is effective within a procedure, but not effective in optimizing iterative execution of procedures containing queries. In general, cases where a query execution is deeply nested within a procedure call chain with a loop in the outermost procedure, are quite common in database applications, and they restrict the applicability of asynchronous submission and batching. Such cases are found especially in applications that use object relational mapping tools such as Hibernate.

Consider a query which is executed in procedure M (like the *computePartCount* procedure in Example 3), which is invoked from within a loop in procedure N. Performing loop fission to enable batching (or asynchronous submission) requires one of the following transformations to M: (i) a set-oriented (or asynchronous) version of M, (ii) fission of procedure M into two at the point of query execution, (iii) inlining of M in N. All these transformations are very intrusive and complex.

A more elegant solution would be to issue an asynchronous request for the query in advance (in procedure N). Once a prefetch request for the query is placed directly within the loop, the loop can be transformed to enable batching or asynchronous submission as described earlier. Manually identifying the best points in the code to perform prefetching is hard due to the presence of loops and conditional branches; it is even harder in the presence of nested procedure invocations. Manually inserted prefetching is also hard to maintain as code changes occur.

A technique to automatically insert prefetch requests for queries at the earliest possible points in a the

Example 5 Chaining and Rewriting prefetch requests for Example 3

```
submitChain("select category_id from categories where cat_group=?",
        "select count(partkey) as part_count from part where p_category=?", group_id, "q1.category_id")
    // the program remains unchanged as in Example 3
```

Example 6 The final program of Example 3 after loop fission

```
rs = executeQuery("select category_id from categories where cat_group=?", group_id);
while(rs.next()) {
    category = rs.getInt("category_id");
    bstmt.addBatch(category);
}
bstmt.submitBatch();
for(LoopContext ctx: lct) {
    category = ctx.getInt("category");
    partCount = computePartCount(category); // code for this function remains as in Example 3
    sum += partCount;
}
```

program across procedure calls is presented in [18]. In general, the goal of prefetching is to insert asynchronous query requests at the earliest possible points in the program so that the latency of network and query execution can be maximally overlapped with local computation. Suppose a query q is executed with parameter values v at point p in the program. The earliest possible points e where query q could be issued are the set of points where the following conditions hold: (a) all the parameters of q are available, (b) the results of executing q at points e and p are the same, and (c) conditions (a) and (b) do not hold for predecessors of e. For efficiency reasons, we impose an additional constraint that no prefetch request should be wasted. In other words, a prefetch request for query q with parameters v should only be inserted at earliest points where it can be guaranteed that q will be executed subsequently with parameters v.

Detecting earliest possible points for queries in the presence of multiple query execution statements, while satisfying the above constraints, requires a detailed analysis of the program. The presence of conditional branching, loops and procedure invocations lead to complex interstatement data and control dependences which are often not explicit in the program. We approach this problem using a data flow analysis framework called *anticipable expressions analysis* and extend it to compute *query anticipability* [18].

The transformed program after inserting the prefetch request in the calling procedure of Example 3 is shown in Example 4. The prefetch request for query in the procedure *computePartCount* is placed directly in the loop just before the procedure invocation. The *submitQuery* API is a non-blocking call that submits the query to a queue and returns immediately. The queue is monitored by a thread pool as done for asynchronous submission (Section 3). Note that the procedure *computePartCount* remains unchanged. This is because when the prefetch request initiated by *submitQuery* completes, the results are placed in a cache. The *executeQuery* API within the *computePartCount* procedure will first look up this cache, and block if the results have not yet arrived.

Chaining and rewriting prefetch requests: A commonly encountered situation in practice is the case where the output of one query feeds into another, such as the case in Example 4. This is an example of a *data dependence barrier* [18], where the dependence arises due to another query. For example say a query $q1$ forms a barrier for submission of $q2$, but $q1$ itself has been submitted for prefetch as the first statement of the method. As soon as the results of $q1$ become available in the cache, the prefetch request for $q2$ can be issued. This way of connecting dependent prefetch requests is called chaining. In DBridge, this is implemented using the *submitChain* API, and for our example, the *submitChain* invocation is shown in Example 5.

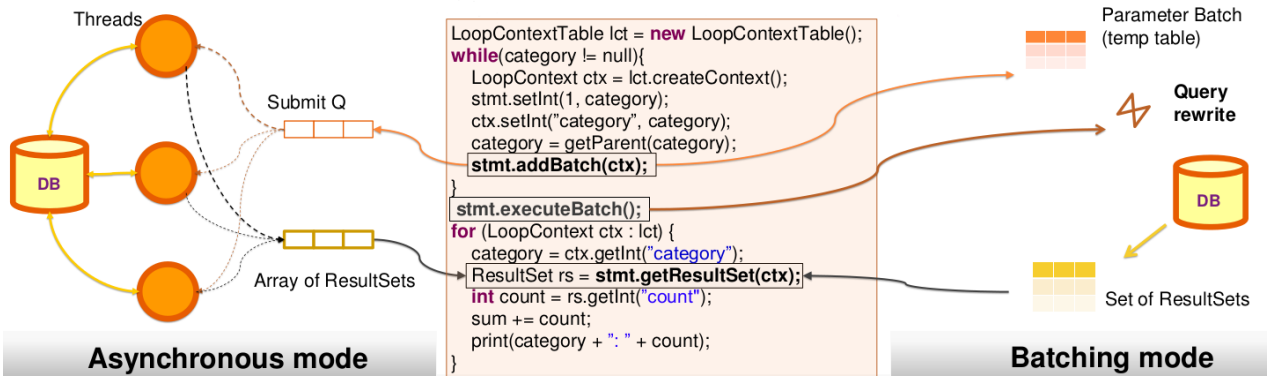


Figure 1: DBridge Batching and Asynchronous query submission API

Chaining by itself can lead to substantial performance gains, especially in the context of iterative query execution whose parameters are from a result of a previous query. Chaining collects prefetch requests together, resulting in a set of queries with correlations between them. Such queries can be combined and rewritten using known query decorrelation techniques [9]. In order to preserve the structure of the program, the results of the merged rewritten query are then split into individual result sets and stored in the cache according to the individual queries. In our implementation, the *submitChain* API internally performs query rewriting.

Integration with loop fission: Observe that in Example 4, we could alternatively perform batching or asynchronous submission by loop fission, as described in Sections 2 and 3. The rewritten program would then look as shown in Example 6. If the loop is over the results of a query, and the result attribute(s) of the query feed into the query within the loop, rewriting using the *submitChain* API is more beneficial since it achieves set oriented execution while avoiding the construction of a temporary batch table. Also, the *submitChain* API is less intrusive to the original program. The loop fission approach is beneficial in cases where the preconditions for *submitChain* [18] are not satisfied.

5 System Design and Implementation

We have implemented the above techniques and incorporated them into DBridge [4, 17], a tool that optimizes Java programs that use JDBC. The techniques however, are general, and can be adapted to other languages and data access APIs. Our system includes two components (i) a source-to-source program transformer, and (ii) a runtime batching and asynchronous submission framework. We now briefly describe these components.

Program Transformer: The program transformations are identical for both batching and asynchronous query submission. The analyses and the transformation rules have been built using the SOOT optimization framework [19], with Java as the target language and JDBC as the database access API. SOOT uses an intermediate code representation called Jimple and provides dependency information on Jimple statements. Our implementation analyzes and transforms Jimple code and finally, the Jimple code is translated back into a Java program.

Runtime Batching/Asynchronous Submission Framework: The DBridge runtime library works as a layer between the actual data access API and the application code. In addition to wrapping the underlying API, this library provides batching and asynchronous submission functions, and manages threads and caches. The library can be configured to either use batching, asynchronous submission, or asynchronous batching.

Set oriented execution API: The right side of Figure 1 shows the behaviour of the DBridge library in the batching mode. The first loop in the transformed program generates all the parameter bindings. Next, a rewritten form of the query is executed to obtain results for all the parameter bindings together. Then, the second loop executes statements that depend on the query results. Query rewrite is performed at runtime within DBridge's

implementation of the *executeBatch* method, which internally transforms the query statement into its set oriented form, as described in Section 2.

Asynchronous Query Submission API: The left side of Figure 1 shows the behaviour of the asynchronous submission API. The first loop in the transformed program submits the query to a queue in every iteration by invoking the *stmt.addBatch(ctx)* function. The queue is monitored by a thread pool which manages a configurable number of threads. The requests are picked up either individually or in batches, by free threads which maintain open connections to the database. The threads execute the query in a synchronous manner i.e., they block till the query results are returned. The results are then placed in a cache keyed by the loop context(ctx). In the second loop, the program invokes *getResultSet*, which is a blocking function. It first checks the cache if the results are already available, and if not, blocks till they become available.

6 Related Work

In the past, database researchers have explored the use of program analysis to achieve different objectives. Early approaches presented in [1, 12, 8, 13] combine program analysis and transformations in different ways to improve performance of database applications. Recently Manjhi et al. [14] describe program transformations for improving application performance by query merging and non-blocking calls. Chaudhuri et al. [2, 3] propose an architecture and techniques for a static analysis framework to analyze database application binaries that use the ADO.NET API. There has also been recent work on inferring SQL queries from procedural code using program synthesis by Cheung et al. [7, 6]. In this article, we present a consolidated summary of our work [11, 5, 18, 16] on program optimizations enabled by static analysis of the code. The techniques presented are applicable for general purpose programming languages such as Java, with embedded queries or Web service calls.

7 Open Challenges

Although many approaches and techniques for database aware program optimizations have been proposed, there are more opportunities that remain to be explored. We now discuss some open challenges and directions for future work in this area.

The techniques we have described in this paper are purely based on static analysis. There have been other approaches that use logs, traces and other runtime information for optimization. An interesting area to explore is a combination of these complementary approaches to achieve more benefits. Also, in this paper we have described techniques in the context of interactions between an application and a database. However these techniques are more general and can be extended to optimize interactions in other client server environments. Consider applications running on mobile devices or Web browsers. They interact with services typically over HTTP. These interactions are currently manually optimized using techniques similar to prefetching and batching optimizations. Adapting our techniques to automate these scenarios is an interesting and important area.

8 Conclusions

Taking a global view of database applications, by considering both queries and imperative program logic, opens up a variety of opportunities to improve performance. To harness such opportunities, program and query transformations must be applied together. Many program analyses and transformations well known in the field of compilers can profitably be made use of to optimize database access. In this article, we have given an overview of various techniques to optimize database applications, and also described some directions in which this work can be extended. We believe that this problem has the potential to attract more interest from both the database and the compilers community in future.

References

[1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. Comput.*, 30(5):341–356, 1981.

[2] S. Chaudhuri, V. Narasayya, and M. Syamala. Bridging the application and DBMS divide using static analysis and dynamic profiling. In *SIGMOD*, pages 1039–1042, 2009.

[3] S. Chaudhuri, V. Narasayya, and M. Syamala. Database application developer tools using static analysis and dynamic profiling. In *IEEE Data Engineering Bulletin Vol 37, No 1*, March 2014.

[4] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. DBridge: A program rewrite tool for set-oriented query execution. In *ICDE*, pages 1284–1287, 2011.

[5] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *ICDE*, 2011.

[6] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. In *IEEE Data Engineering Bulletin Vol 37, No 1*, March 2014.

[7] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. PLDI '13, pages 3–14, New York, NY, USA, 2013.

[8] G. B. Demo and S. Kundu. Analysis of the Context Dependency of CODASYL FIND-Statements with Application to a Database Program Conversion. In *ACM SIGMOD*, pages 354–361, 1985.

[9] C. A. Galindo-Legaria and M. M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *ACM SIGMOD*, 2001.

[10] R. Guravannavar. *Optimization and Evaluation of Nested Queries and Procedures*. Ph.D. thesis, Indian Institute of Technology, Bombay, 2009.

[11] R. Guravannavar and S. Sudarshan. Rewriting Procedures for Batched Bindings. In *Intl. Conf. on Very Large Databases*, 2008.

[12] R. H. Katz and E. Wong. Decompiling CODASYL DML into Retional Queries. *ACM Trans. on Database Systems*, 7(1):1–23, 1982.

[13] D. F. Lieuwen and D. J. DeWitt. A Transformation Based Approach to Optimizing Loops in Database Programming Languages. In *ACM SIGMOD*, 1992.

[14] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic Query Transformations for Dynamic Web Applications. In *ICDE*, 2009.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[16] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. Program transformations for asynchronous and batched query submission. *CoRR*, abs/1402.5781, January 2014.

[17] K. Ramachandra, R. Guravannavar, and S. Sudarshan. Program analysis and transformation for holistic optimization of database applications. In *Proc. of the ACM SIGPLAN SOAP*, pages 39–44, 2012.

[18] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *SIGMOD*, pages 133–144, 2012.

[19] Soot: A Java Optimization Framework: http://www.sable.mcgill.ca/soot.

Abstraction Without Regret in Database Systems Building: a Manifesto

Christoph Koch, EPFL

Thoughts on joint work with:

Yanif Ahmad[+], Hassan Chafi[**], Thierry Coppey[*], Mohammad Dashti[*], Vojin Jovanovic[*], Oliver Kennedy[#], Yannis Klonatos[*], Milos Nikolic[*], Andres Noetzli[*], Martin Odersky[*], Tiark Rompf[*,**], Amir Shaikhha[*]

[*]EPFL [**]Oracle Labs [+]Johns Hopkins University [#]SUNY Buffalo

Abstract

It has been said that all problems in computer science can be solved by adding another level of indirection, except for performance problems, which are solved by removing levels of indirection. Compilers are our tools for removing levels of indirection automatically. However, we do not trust them when it comes to systems building. Most performance-critical systems are built in low-level programming languages such as C. Some of the downsides of this compared to using modern high-level programming languages are very well known: bugs, poor programmer productivity, a talent bottleneck, and cruelty to programming language researchers. In the future we might even add suboptimal performance to this list. In this article, I argue that compilers can be competitive with and outperform human experts at low-level database systems programming. Performance-critical database systems are a limited-enough domain for us to encode systems programming skills as compiler optimizations. However, mainstream compilers cannot do this: We need to work on optimizing compilers specialized for the systems programming domain. Recent progress makes their creation eminently feasible.

HAVE THE CAKE AND EAT IT TOO. *Abstraction without regret* refers to high level programming without performance penalty [RO12, Koc13]. The **main thesis of this article** is that databases implemented in high-level programming languages can beat state-of-the-art databases (implemented in C) performance wise.

MONUMENTS TO ABSTRACTION FAILURE. Let us look at the core of a classical SQL DBMS. On a high level of abstraction, textbook-style, we see nice, cleanly separated components, such as a storage manager, a buffer manager, a concurrency control subsystem, and a recovery manager. On a high level, the interfaces between these components are relatively clear, or are they?

In practice, the code of this core is a great monolith, formidable in its bulk (millions of lines of code). Buffer and storage management are tightly integrated. The page abstraction is a prime example of abstraction failure, and stores identifiers relevant to other subsystems, such as recovery. Concurrency control rears its ugly head when it comes to protecting ACID semantics against insertions and deletions, and when there are multiple access paths to data, some hierarchical like B-trees. In practice, concurrency control code is interspersed throughout

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

the core. It is hard to even place concurrency control algorithms in their separate source code files; much is inlined in source files of the page abstraction, B-trees, and such. Physical logging breaks the abstractions of pretty much each of these subsystems. Sadly, this paragraph could be continued much longer.

WHY THE MONOLITH TODAY? The existence of the Monolith is a consequence of the use of C for building such systems. Modern programming languages offer features to create abstractions and protect them from leaking. We have powerful type systems and a strong arsenal of software composition primitives at our disposal, such as modules, interfaces, OO features (classes, objects, encapsulation, etc.), and genericity. Design patterns show us how to use these language features to solve abstraction problems and compose software in a maintainable way. Can modern high-level programming languages disentangle the Monolith and reclaim clean abstractions and interfaces? Of course: As Butler Lampson (and earlier David Wheeler) [Lam93] has claimed, *any problem in computer science can be solved by an additional level of indirection*. In the context of structuring software artifacts and eliminating abstraction leaks, this statement is true.

The problems come when performance matters. As Lampson's law suggests, these modern software composition concepts increase indirection, which compromises performance. It is the thesis of this article that compilers can eliminate these indirections automatically and reclaim efficiency; so we database people can dismantle the Monolith and do not need to keep living in PL stone age (pun intended).

BENEFITS OF HIGH-LEVEL PROGRAMMING. The main benefit of high-level programming languages is much increased productivity. The same functionality tends to be implementable in fewer lines of code[1], which improves maintainability and agility of software development. The improved productivity results in part from the fact that advanced language features such as genericity allow for more reusable code. Software libraries have better coverage of the needs of programmers than do libraries of low-level languages. Programmers approach software development with a different mindset: Resigned from the hope of the last bit of efficiency when using a high-level language, they reuse library code even when a re-implementation might yield slightly better performance. For example, in building systems in C with an emphasis on efficiency, we often feel the urge to build that additional list or tree data structure from scratch. In high-level programming, elegant generic collection classes readily available are just too attractive, and this rarely happens.

High-level programming languages, particularly functional ones, also tend to be better suited for automatic program analysis and transformation because analysis is localized and search spaces are smaller. One notable reason why this is true is that code that follows style conventions avoids hard-to-analyze counter arithmetics for core tasks such as looping through data structures, and uses abstractions such as map and foreach instead. Of course, it is much easier to robustly fuse or reorder foreach loops than C-style for loops with a loop counter. As a consequence, compilers can perform more powerful optimizations and transformations on high-level programs than on low-level ones. Fortunately, we can conclude from successful recent programming language designs such as Python and Scala that modern high-level programming languages are functional. Even Java is currently becoming a functional language [Jav13].

SUCCESS STORIES. Some recent successes of high-level programming in systems include the Singularity operating system from Microsoft Research [LH10] and Berkeley's Spark [ZCD+12]. Singularity is an operating system implemented almost in its entirety in a dialect of C#, and runs as managed code in Microsoft's CLR virtual machine. Still it outperforms other operating systems that are written in highly optimized C code. This is achieved not by a superior compiler but by program analysis and transformation techniques that would not be possible in C. For example, the source code of processes is analyzed for their potential for safe merging: such *software-isolated processes (SIPs)* are protected from each other using programming language constructs such as types only, but are not isolated by the operating system at runtime. A group of merged SIPs runs as a

[1] The folklore has it that the improvement can be by up to two orders of magnitude, see also [Pyt14, Lea13].

single process, and the performance benefit stems from the reduced amount of process switching, which is very expensive in classical operating systems.

Spark provides (fault-tolerant) distributed collections in Scala. It is easy to build map-reduce like systems, with improved performance compared to Hadoop and a much smaller code base (14000 lines of code [ZCD+12]). This is an example of how the improved productivity of high-level languages frees developer's cycles, which can be channeled into better designs. Spark benefits from Scala's functional nature by using closures, which are serialized and stored (for logical logging) as well as used to ship work between nodes.

These are examples of how high-level optimizations can dominate the cost of high-level programming. This should not come as a surprise; it is not foreign to database experience. Take the field of query processing. Optimizing join orders matters more than the choice of query operators. Algorithmic choices, such as choosing a good join operator, matter more than low-level implementation decisions. The choice of programming language generally has just a constant factor impact on performance. Performance is dominated by algorithmic and architectural decisions.

COSTS OF HIGH-LEVEL PROGRAMMING. Complex software implemented in high-level programming languages makes extensive use of composition from smaller components. This includes components from previously existing libraries and components developed specifically as part of the new software artifact. Clean designs lead to deep nesting of class and interface hierarchies (both in terms of aggregation and inheritance/subtyping). Objects are copied rather than referenced to protect encapsulation. Abstraction requires complex and deep aggregations of containers, and thus the frequent creation, copying, and destruction of objects at runtime. Even basic types such as integers get boxed in objects for use in generic containers. Function call stacks are very deep, with correspondingly high overheads due to the creation of activation records and such[2]. Depending on how classes and generic code are instantiated, large amounts of code may be duplicated; much library code remains unused at runtime[3]. Functional programming languages in particular increase the amount of auxiliary object creation and destruction (boxing and unboxing) to impressive levels. Naïve compilers and runtime systems may see hundreds of object creations and destructions for a single constant-time call in a rich library[4]. Of course, modern (just-in-time) compilers such as Java Hotspot use advanced techniques to reduce boxing and unboxing, and runtime systems are optimized to handle the objects that cannot be avoided with relatively little overhead.

THE USE OF LOW-LEVEL LANGUAGES IN SYSTEMS BUILDING. Truth be told, compilers that eliminate all abstraction costs of developing a DBMS in a high-level programming language are not readily available today, but can be built, and with less effort than a classical DBMS. I will discuss how below.

The absence of suitable compilers is not a conclusive explanation for the use of C. If we started building a DBMS from scratch today, the rational approach would be to do it in a high-level language, and leverage the gained productivity to implement high-level performance-enhancing ideas first. If we ever run out of such ideas and converge to a stable architecture and feature set, we may decide to lower and specialize our code for optimal performance – but we would want to do this with the help of tools.

The reason for the use of C is in part historical. The first relational DBMS used C because it was the state of the art in programming languages for which good compilers were available. We never rethought our modus operandi sufficiently since. Today, the major players are invested in very large C codebases and armies of C programmers strongly bonded to their codebases. They have little choice but to continue.

Even in academic circles, building systems prototypes in C today is a matter of pride and not entirely rational.

[2]Compiler remedy: inlining.

[3]Compiler remedies: common subexpression elimination (CSE) and dead code elimination (DCE).

[4]Compiler remedy: depending on context, inlining, fusion, and deforestation.

THE BAR IS LOW. Isn't the correctness of the thesis of this article unlikely, given that the major commercial database systems are efforts of tens of thousands of expert years, and thus highly refined and efficient? Even in their advanced states, these code bases are in flux; feature requests keep coming in and algorithmic and architectural improvements keep happening, preventing the code bases from converging to that hypothetical state of perfect, stable maturity that can just not be beaten performance-wise. It has been argued, though, that the current DBMS codebases are rather elderly and tired anyway [SMA+07]. A new development from scratch, no matter whether in a high-level or low-level PL, would look very different.

More fundamentally, there are multiple reasons why even low-level DBMS source code must make ample use of indirection. Indirection (using procedures etc.) is necessary for basic code structuring, to keep the codebase from exploding into billions of lines of code. Manually inlining such indirections would make it impossible for humans to maintain and work with the codebase. Here, the human element of software development is in conflict with that last bit of performance a compiler might obtain. To illustrate the tradeoff between indirection and inlining, in a recent version of Postgres, there were seven B-tree implementations and at least 20 implementations of the page abstraction [Klo12], variously inlined with other behavior. This was certainly done for performance, which could be improved even further by more inlining, which becomes impractical to do manually.

A DBMS that implements the ANSI three-layer model needs indirection to realize the layering, for instance to support a dynamically changeable schema. Here, indirection is a *feature* of the system, but its performance-degrading effects could be eliminated by *staged compilation*, that is, *partial evaluation* to create code just-in-time that hardwires the current schema and eliminates schema-related indirections (cf. [RAC+14] for an example). This is easy to do using frameworks such as MetaOCAML [Met] or LMS [RO12], but impractical for C.

Thus the bar is low: There are many angles of attack by which a DBMS implemented in a high-level PL may outperform state-of-the-art DBMS.

DOMAIN-SPECIFIC LANGUAGES. Let us consider systems code written in a high-level imperative (such as Java) or impurely[5] functional programming language (such as Scala). There is an important special case that is worth discussing. Suppose we restrict ourselves to using dynamic datastructures only from an agreed-upon, limited library. A priori, there are no restrictions regarding which programming language features (and specifically control structures) may be used, but we agree not to define further dynamic data structures, avoid the use of imperative assignments other than of values of the provided data structures, and use collection operations such as foreach, map, fold, and filter to traverse these data structures rather than looping with counters or iterators. We can call such a convention a *shallowly embedded*[6] *domain-specific language (DSL)*. Natural choices of data structures to create a DSL around would be, say, (sparse) matrices and arrays for analytics, or relations for a PL/SQL style DSL. Thus, the unique flavor of the DSL is governed by the abstract data types it uses, and we do not insist on new syntax or exotic control structures.

AUTOMATICALLY ELIMINATING INDIRECTION. The relevance of the restriction to such a DSL will become apparent next. Let us look at which compiler optimizations we need to eliminate the indirections resulting from high-level programming discussed above. Some optimizations, such as duplicate expression elimination and dead code elimination, are generic and exist in some form in many modern compilers.

However, other key optimizations depend on the dynamic data structures. Most loops in programs are over the contents of collection data structures, so let us consider optimizing this scenario. Transforming a collection causes the creation and destruction of many objects; the consecutive execution of two such transformations

[5] That is, which supports imperative programming when necessary.

[6] Shallow embedding means that the language extension is made through a library; no lifting of the language into an abstract syntax tree is required to be able to process the DSL using a mainstream programming language and compiler.

causes the creation of an intermediate collection data structure. Such intermediate object creations and destructions can in principle be eliminated by loop fusion and deforestation.

Let us consider the case that we have an agreed upon interface for collections with a higher-order function map(f) that applies a function f to each element of the input collection c and returns the collection of results of f. Then we can fuse and deforest c.map(f).map(g) into c.map(g ∘ f). The optimized code performs a single traversal of the collection and produces the collection of results of the composed function g ∘ f applied to each member of c. The original code is less efficient because it in addition creates and destroys an intermediate collection c.map(f) along the way. Of course this transformation is only permitted if f and g have no side effects. In addition, the interface of collections must be known to the compiler. This example is admissible both for functional and imperative collections. However, particularly for the imperative collection implementations, the admissibility of this fusion optimization is too hard to determine automatically (by program analysis) if the compiler does not *know of collections*.

DOMAIN-SPECIFIC VS. GENERIC COMPILER OPTIMIZATIONS. For every possible transformation, a compiler implementor has to make a decision whether the expected benefit to code efficiency justifies the increase in the code complexity of the compiler. Optimizations that rarely apply will receive low priority.

The core of a collections interface using an operation map(f)[7] is known as a monad [BNTW95]. Monads play a particularly important role in pure functional programming languages such as Haskell, where they are also used to model a variety of impure behaviors such as state and exceptions, in addition to collections [Wad95]. The frequent occurrence of monads calls for compilers to be aware of them and to implement optimizations for them. Thus Haskell compilers implement fusion and deforestation optimizations for monads.

In compilers for impure and imperative languages, however, monads are unfortunately still considered exotic special cases, and such optimizations are generally not implemented.[8] This motivates DSLs constructed around abstract data types/interfaces on which algebraic laws hold that call for special optimizations. If the DSL is of sufficient interest, a compiler for this DSL can be built that supports these optimizations.

AUTOMATIC DATA STRUCTURE SPECIALIZATION. Once we decide to fix a DSL and build an optimizing compiler for it, it is natural to add further intelligence to it. One natural thing to do is to make the DSL only support one intuitively high-level data structure and to make it easy to use by creating a high-level interface. Performance can then be gained by automatically specializing the data structure implementation in the compiler, decoupling the user's skill from the efficiency of the code. For example, a DSL may only support one high-level matrix data structure, and internally specialize it to a variety of low-level implementations and normal forms. A PL/SQL style DSL may offer a single relation abstraction, forcing the user to program on a high level without reference to index structures and such; the compiler may then introduce a variety of row, columnar, or other representations, as well as indexes. Of course, this separation of logical and physical data representation is a key idea in relational databases and very well studied. What compilers contribute is the potential elimination of all indirection overhead arising from this separation of concerns. In a main-memory database scenario, we deal with lightweight accesses to in-memory structures that can be inlined with the workload code, while an access to a secondary-storage (index) structure may turn into the execution of millions of CPU instructions.

EVIDENCE: THE CASE OF TPC-C. Above, we have covered two paths to performance (basic compiler optimizations and data structure specialization/index introduction), and discussed how a DSL compiler can implement both. But are these two groups of code transformations and optimizations really sufficient to do

[7] This is imprecise in many ways, but because of space limitations, I just refer to [Wad95, BNTW95].

[8] Programmers are assumed to still prefer looping with iterators or counters over using such interfaces. This will change with time as a new generation of programmers grows up with frameworks such as LINQ [Mei11], map/reduce, and Spark, which impose monad-style interfaces.

the job? In [SMA+07], the H-Store team went through an exercise that suggests the answer is yes. The paper describes an experiment in which expert C programmers implemented the five TPC-C transaction programs in low-level C, using all the tricks of their repertoire to obtain good performance. This code is used as a standard of efficiency in the authors' argument for a new generation of OLTP systems, and outperforms classical OLTP systems (which however have additional functionality[9]) by two orders of magnitude. An inspection of the resulting code [DCJK13] shows that the expertly written C code can be obtained by fusion, deforestation, and inlining, plus data structure specialization[10] from a naïve implementation in a high-level DSL of the type of PL/SQL. The paper [DCJK13] presents a DSL compiler that implements exactly these optimizations. The compiler generates low-level code from PL/SQL implementations of the TPC-C transaction programs. This code outperforms the hand-written C code of [SMA+07] but would have been equivalent if the programmers of [SMA+07] had applied their optimization ideas consistently and exhaustively.

THE KEY MANTRA OF DSL PERFORMANCE. This has been observed time and again in research on DSL-based systems [BSL+11, RSA+13, DCJK13, KRKC14]:

The central abstract data type shall be the focal point of all domain-specific optimizations.

If we consider that, in the context of RDBMS, this just means *mind the relations*, it is really obvious. But the meaning of the mantra is this: Computations do not happen in a vacuum: They happen on data, and the bulk of this data is stored in values of the key abstract data type (relation, matrix, etc.). Such a value is not atomic; computations loop intensively over it, and in optimizing these loops (most of them of the map, fold, and filter varieties) lies the greatest potential for code optimization.

REVISITING THE SQL DSL. SQL is the most successful DSL yet, and the database community has amassed a great deal of expertise in making SQL-based DBMS perform. Leaving aspects of concurrency and parallelization aside, SQL databases draw their performance from (1) keeping the search space for optimizations under control and (2) leveraging domain-specific knowledge about an abstract datatype of relations (which ultimately governs representation, indexing, and out-of-core algorithms). We can relate (1) to the *functional* and *domain-specific* (and thus concise) nature of relational algebra.

Since SQL supports a universal collection data type (relations) and language constructs for looping over (FROM/joins) and aggregating the data, SQL systems can be used to embed many other DSLs by moderate extension. Consequently, we have studied SQL extended by UDFs, objects, windows, cubes, counterfactuals, PL control structures, probabilities, simulations, new aggregates for analytics, machine learning primitives, and many other features. By our mantra, SQL engines make a nontrivial contribution to the optimized execution of these DSL embeddings. However, in general, embedding a DSL into an SQL system in this way causes us to miss optimization opportunities unless we make use of an extensible optimization framework that allows us to instruct the system how to exploit additional algebraic properties of the DSL. This could be an extensible query optimizer (cf. [HFLP89]) or an extensible DSL compiler framework such as EPFL's LMS [RO12, RSA+13], Stanford/EPFL's Delite [BSL+11] and Oracle's Graal/Truffle [WW12]. These projects provide infrastructure to easily and quickly build DSL compilers and efficient DSL-based systems.

DATABASE (QUERY) COMPILERS IN THE LITERATURE. The compilation of database queries has been studied since the dawn of the era of relational DBMS. IBM's System R initially used compilation techniques for query execution [CAB+81] before, even prior to the first commercial release of the system, converging to the now mainstream interpretative approach to query execution. Recent industrial data stream processing

[9] Both the C code of [SMA+07] and the code generated by the compiler of [DCJK13] run in main memory and there is no concurrency control, latching, or logging.

[10] This comes in two forms, specialization of relations to arrays for tables that cannot see insertions or deletions during the benchmark, and the introduction of (B-tree) indexes to efficiently support certain selections in queries.

systems such as Streambase and IBM's Spade also make use of compilation. Microsoft's Hekaton [DFI+13] compiles transaction programs.[11] Furthermore, there have been numerous academic research efforts on compiling queries; recent ones include [RPML06], DBToaster [AK09, AKKN12], [KVC10], and Hyper [Neu11].

It is fair to say, though, that compiling queries is not mainstream today. The reason for this is that we have typically made our lives very hard for ourselves when creating compilers. System R abandoned compilation because it was done by simple *template expansion*, causing a productivity bottleneck [Moh12]. Rather than implementing an algorithm A, one had to implement an algorithm B that, when called, would generate the code of algorithm A – ultimately by string manipulation. This substantially slowed down development in a time when little was established about building RDBMS and much had to be explored. Code generation by template expansion of query operators is still practiced although decades behind the state of the art in compilers.[12]

We do not put enough effort into designing our compilers well. Often, building a compiler is a subservient goal to building a DBMS, and the compiler's design ends up being a hack. Focusing on just one DSL that has rather limited structure (queries) makes it hard to get our compilers right. Often, our compilers grow organically, adding support for new language constructs as the need arises or we discover a previous design to be flawed. With time this leads to unmanageable code that is ultimately abandoned.

This calls for a separation of concerns. A compiler should be developed for a well-defined general-purpose language, ideally by compiler experts; it can later be customized for a DSL.[13] In this article, I am arguing for compiling entire systems, not just queries. Thus the compiler must support a sufficiently powerful language. By aiming to build extensible compilers for general-purpose programming languages, we may be spreading ourselves too thin; fortunately, such compiler frameworks are now readily available [RO12, BSL+11].

Two observations regarding modern compiler technology that are valuable for people interested in building database compilers concern *lowering* and modern *generative programming*.

LOWERING. It is considered good practice to build a compiler that transforms intermediate representations (IRs) of programs by lowering steps, which transform IRs to lower and lower levels of abstraction (cf. e.g. [RSA+13]). Ideally, the final IR is at such a low level of abstraction and conceptually so close to the desired output code that code generation reduces to simple stringification. Compared to template expansion where DSL operations are replaced by monolithic code templates, breaking up the compilation process into smaller steps creates additional optimization potential and allows the type system of the PL in which the compiler is developed to guide the development process.

MODERN GENERATIVE PROGRAMMING. There is a rich set of ideas for making compiler implementation easier, from quasiquotes [TS00] to tagless type-based generative metaprogramming systems [CKS09, RO12] and multi-staging [Tah03]. This work makes use of the concept of compilation as *partial evaluation* in various ways (cf. [RAC+14] for an illustration of partial evaluation applied to compiling away the data dictionary).

Lightweight Modular Staging (LMS) [RO12] is a framework for easily creating optimizing compilers for DSLs. It is written in Scala and offers a Scala compiler as a library. Using Scala's powerful software composition features, it is easy to add domain-specific optimizations to this compiler. Scala-based DSLs do not even require the creation of a parser, because of Scala's ability to lift Scala code into an abstract syntax tree using language virtualization. LMS offers default code generators for Scala and C, and the Delite project [BSL+11] offers a parallelizing collections library and code generators for map/reduce and CUDA. In addition, the Delite team

[11] Compared to [DCJK13], Hekaton performs less aggressive inlining (also with a view towards code size) and puts less emphasis on removing indirection. It mostly preserves relational plan operator boundaries and wires operators together by gotos [FIL14].

[12] Citations elided out of courtesy: the reader is invited to inspect the recent papers cited in the previous paragraph.

[13] Take LLVM/clang, which was or is used in a number of query compilation projects [AK09, Neu11, WML14, VBN14]. The clang compiler was done by compiler experts, but it is not easy to extend by domain-specific optimizations. Typically, code given to clang is too low-level for most of the collection optimizations discussed earlier [KRKC14].

have created a growing number of DSLs (for querying, graph analytics, machine learning, and mesh computing) together with LMS-based compilers.

Staging in LMS refers to support of generative metaprogramming where representation types are used to schedule partial evaluation at different stages in the code's lifecycle. This allows to easily create programming systems that offer arbitrary combinations of static compilation, interpretation, and just-in-time compilation [RSA+13]. The usefulness of generative metaprogramming with LMS in the context of data management is illustrated in [RAC+14] using an example very similar to eliminating indirections due to a data dictionary in a just-in-time compiler. Using LMS, optimizing DSL compilers can typically be written in 3000 lines of code or less [BSL+11, AJRO12, DCJK13, KRKC14].

COMPILING SYSTEMS WITHOUT REGRET. There is increasing evidence that compilers can match and outperform human expert programmers at creating efficient low-level code for transaction programs [DCJK13] and analytical queries [KRKC14].[14] Admittedly, this does not yet "prove" the main thesis of this paper for complete database systems that deal with multi-core, parallelization, concurrency, and failures. Here, however, I point at systems such as Singularity and Spark that address such aspects and are implemented in high-level languages, with convincing performance. Ultimately, implementations of the aspects of systems that I have mainly brushed aside in this article spend much of their time in system calls. They depend less on low-level code efficiency and more on the choice of protocols and algorithms. Using a high-level PL, even unoptimized, is not much of a disadvantage [LH10].

An interesting goal for the future is to create a (database) *systems programming DSL*, adapting the ideas of work like [BGS+72] to the current millenium's changed perceptions of what constitutes high-level programming.

COMPILER-SYSTEM CODESIGN. There are now DSL compiler frameworks such as LMS that are easy to extend by (domain-)specific optimizations. Should we come to face a scenario in which our compiler produces locally bad code for our DBMS, we can always specialize our compiler by plugging in an ad-hoc transformation that deals with that particular issue at this place in the code. A general solution of the issue may not be easy to come by, but such a hack is. In terms of productivity, we profit from overall development of the DBMS in a high-level language and the fact that such hacks will be rare and will be needed in at most a few particularly performance-critical places in the codebase. We can eulogize such a strategy as *compiler-system codesign*. Isn't this a remedy to *any* unexpected obstacle to the truth of this article's thesis that we may yet encounter?

ONE FACTORY FITS ALL. Some researchers have criticized the commonplace "one size fits all" attitude in databases [Sto08] and have argued for the specialization of DBMS to render them well-matched to their workloads. Elsewhere, the argument is taken to call for the creation of one specialized system from scratch for each relevant use case. We should reconsider this and let compilers, rather than humans, do the leg work of specialization. Given a suitable library of DBMS components (query operators, concurrency control algorithms, etc.), there can be a single compiler-based factory for generating all these specialized database systems.

References

[AJRO12] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded DSL for high performance big data processing. In *International Workshop on End-to-end Management of Big Data*, 2012.

[14] In [KRKC14], we implement an analytical query engine in Scala and use our LMS-based optimizing compiler to compile *it together with a query plan* to remove abstraction costs and obtain competitive query performance. These two papers are first steps in an ongoing effort at EPFL to build both an OLTP and an OLAP system in Scala to provide practical proof of this article's thesis.

[AK09] Yanif Ahmad and Christoph Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.

[AKKN12] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. In *VLDB*, 2012.

[BGS+72] R. Daniel Bergeron, John D. Gannon, D. P. Shecter, Frank Wm. Tompa, and Andries van Dam. Systems programming languages. *Advances in Computers*, 12:175–284, 1972.

[BNTW95] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[BSL+11] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, pages 89–100, 2011.

[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[DCJK13] Mohammad Dashti, Thierry Coppey, Vojin Jovanovic, and Christoph Koch. Compiling transaction programs. 2013. Submitted for publication.

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD Conference*, pages 1243–1254, 2013.

[FIL14] Craig Freedman, Erik Ismert, and Per-Ake Larson. Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Engineering Bulletin*, 37(1), March 2014.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *SIGMOD Conference*, pages 377–388, 1989.

[Jav13] Java 8: Project Lambda, 2013. http://openjdk.java.net/projects/lambda/.

[Klo12] Yannis Klonatos. Personal communication, 2012.

[Koc13] Christoph Koch. Abstraction without regret in data management systems. In *CIDR*, 2013.

[KRKC14] Yannis Klonatos, Tiark Rompf, Christoph Koch, and Hassan Chafi. Legobase: Building efficient query engines in a high-level language, 2014. Manuscript.

[KVC10] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[Lam93] Butler Lampson. Turing award lecture, 1993.

[Lea13] Graham Lea. Survey results: Are developers more productive in Scala?, 2013. http://www.grahamlea.com/2013/02/survey-results-are-developers-more-productive-in-scala/.

[LH10] James R. Larus and Galen C. Hunt. The Singularity system. *Commun. ACM*, 53(8):72–79, 2010.

[Mei11] Erik Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, 2011.

[Met] MetaOCaml. http://www.metaocaml.org.

[Moh12] C. Mohan. Personal communication, 2012.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[Pyt14] Python Programming Language Website. Quotes about Python, 2014. http://www.python.org/about/quotes/.

[RAC+14] Tiark Rompf, Nada Amin, Thierry Coppey, Mohammad Dashti, Manohar Jonnalagedda, Yannis Klonatos, Martin Odersky, and Christoph Koch. Abstraction without regret for efficient data processing. In *Data-Centric Programming Workshop*, 2014.

[RO12] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[RPML06] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using JVM. In *ICDE*, 2006.

[RSA+13] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Martin Odersky, and Kunle Olukotun. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*, 2013.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[Sto08] Michael Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12), 2008.

[Tah03] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, 2003.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[VBN14] Stratis Viglas, Gavin Bierman, and Fabian Nagel. Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Engineering Bulletin*, 37(1), March 2014.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[WML14] Skye Wanderman-Milne and Nong Li. Runtime code generation in Cloudera Impala. *IEEE Data Engineering Bulletin*, 37(1), March 2014.

[WW12] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *SPLASH*, 2012.

[ZCD+12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

TCDE Chair

Kyu-Young Whang
KAIST
371-1 Koo-Sung Dong, Yoo-Sung Ku
Daejeon 305-701, Korea
kywhang@cs.kaist.ac.kr

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398