Just-in-time compilation for SQL query processing

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

ABSTRACT

Just-in-time compilation of SQL queries into native code has recently emerged as a viable alternative to interpretation-based query processing. We present the salient results of research in this fresh area, addressing all aspects of the query processing stack. Throughout the discussion we draw analogies to the general code generation techniques used in contemporary compiler technology. At the same time we describe the open research problems of the area.

1. INTRODUCTION

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. Traditionally, relational database systems have compiled SQL into an intermediate representation: the query plan. The query plan is composed of physical algebraic operators, all communicating through a common interface: the iterator interface [9]. The query plan is then interpreted through continuous iterator calls in order to evaluate the query. This technique has effectively made SQL a well-optimized, but interpreted, domain-specific language for relational data management, and has formed the highly successful core of query engine design for more than thirty years.

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 best-known such macro-optimizations. An alternate route that had not been frequently explored and has only recently been revived is applying micro-optimizations stemming from the use of compiler technology. 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.

2. TUTORIAL OUTLINE

We present the salient techniques for seamless integration of just-in-time compilation into query engine design.

2.1 Mixing native code with SQL processing

Compiling SQL into native code is not new. Even System R had a primitive form of code generation in its implementation of SQL's ancestor, SEQUEL [3]. Inherent problems of the mechanism, including the high cost of external function calls and the poor portability to different operating systems, resulted in the approach being abandoned in favor of the now traditional interpretation-based approach. Follow-up work proposed that code should be generated for a high-level programming language [7, 8]. Though this alleviated the portability problem, compilation was still a bottleneck and again the approach was abandoned due to its high performance penalty and therefore limited applicability. The next effort was to precompile parts of the query pipeline, *e.g.,* predicates [2] or aggregate functions [23]. Apart from such approaches, however, native code generation for SQL has remained dormant until recently.

2.2 Just-in-time compilation

In interpreted programming languages, user programs are represented either as source files or as byte code: an intermediate representation for a virtual machine or runtime. To move across platforms, one only needs a version of the runtime for that platform. Byte code, being interpreted, is slower than native code. Just-in-time (JIT) compilation rectifies this: the runtime compiles blocks of byte code into native code on demand. The most popular contemporary runtimes are the Java virtual machine and Microsoft's .NET. Work on JIT compilation goes back to LISP and Smalltalk and is today widely available in contemporary scripting languages and managed runtimes (see [1] for a survey).

2.3 Query compilation in managed runtimes

SQL is effectively a JIT-compiled language. The differences to standard programming languages is that we do not interpret programs but queries. It is not surprising that the revival of native code generation for SQL has started from managed runtimes. The first such effort was the Daytona fourth generation language [10] that used on-the-fly code generation for high-level queries. However, it heavily relied on operating system support and was thus not widely applicable. A similar approach appears in SQLite, where an internal virtual machine provides all database functionality [13]; this exemplifies the potential for synergy between virtual machine technology and database systems. Rao *et al.* [25] used the reflection API of Java to implement a query processor for the Java virtual machine. Though limited in its applicability the approach provided the seed for the more general approaches that have recently emerged.

2.4 Tools and techniques

The majority of tools and techniques for JIT compilation have a programming language pedigree. They primarily target imperative and less frequently declarative and/or functional programming languages. Generic and reusable JIT libraries, *e.g.,* libJIT [4], and GNU lightning [6] provide extensible ways to build, compile, and dynamically invoke new code, thus providing the heavy-lifting of JIT compilation. These tools target the developers of virtual machines, but they are applicable at the micro-optimization level for SQL as well. A better solution is to use a language-agnostic compiler infrastructure like LLVM [19]. Approaches for JIT compilation of SQL translate it into a general purpose language like C; or build an SQL frontend for the intermediate representation; or mix both.

2.5 Just-in-time native code generation for SQL

Just-in-time compilation of SQL has been revived recently. Krikellas *et al.* [17] followed a template-based approach to code generation, inspired by C++, and translated SQL to C. The resulting system exhibited performance that surpassed that of established relational technology by orders of magnitude. Next, Neumann [22] proposed a data-centric approach to query processing, departing from the plan-based representation. The results further corroborate the wide applicability of the just-in-time techniques. These two approaches seeded a stream of work. From combining just-in-time compilation with vectorization [27]; to optimizing I/O [29]; to optimizing data structures [15]; to code generation for GPUs [24]; to applying it in managed runtimes like LINQ [20].

3. OUTLOOK

JIT compilation for SQL is a fresh and potentially phase-shifting approach to a core database research topic: query processing. As is the case with any new research area there are quite a few open questions. One underlying assumption is that the compilation cost can be amortized. This cost may be high in dynamic environments. One future work direction is to automatically identify the queries that are good candidates for compilation. Likewise, admission control policies can be used to decide when queries are admitted and evicted from a compiled query pool, with work on intermediate and final result caching [5, 14, 16, 21, 26] being of benefit here. The application of JIT compilation in heterogeneous multicore is also interesting. Krikellas *et al.* [18] took a first step towards this with multithreaded processing on multicore CPUs, though the JIT compiler was mainly a tool and not the objective of the study. Using code generation fits well with work on JIT-compiled approaches to heterogeneous multicore runtimes like OpenCL [28]. One aspect that has not been addressed so far is transaction processing and concurrency control. One can compile the concurrency control primitives themselves in lower-level and more efficient code without compromising integrity using, *e.g.,* hardware transactional memory primitives [11, 12].

4. REFERENCES

[1] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2), 2003.

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

[3] D. D. Chamberlin *et al.* A history and evaluation of System R. *Commun. ACM*, 24, 1981.

[4] A. Demakov. Just-in-time compiler library, 2007.

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

[6] Free Software Foundation. Using and porting GNU lightning, 2008.

[7] J. C. Freytag and N. Goodman. Translating aggregate queries into iterative programs. In *VLDB*, 1986.

[8] J. C. Freytag and N. Goodman. On the translation of relational queries into iterative programs. *ACM Trans. Database Syst.*, 14, 1989.

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

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

[11] L. Hammond *et al.* Transactional memory coherence and consistency. In *ISCA*, 2004.

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

[13] D. R. Hipp, D. Kennedy, and J. Mistachkin. SQLite Database, 2012.

[14] M. G. Ivanova *et al.* An architecture for recycling intermediates in a column-store. In *SIGMOD*, 2009.

[15] T. Kissinger *et al.* QPPT: Query Processing on Prefix Trees. In *CIDR*, 2013.

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

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

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

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

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

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

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

[23] J. Patel *et al.* Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. In *SIGMOD*, 1997.

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

[25] J. Rao *et al.* Compiled query execution engine using jvm. In *ICDE*, 2006.

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

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

[28] R. Tsuchiyama *et al. The OpenCL Programming Book*. Fixstar, 2012.

[29] Y. Zhang and J. Yang. Optimizing I/O for Big Array Analytics. *Proc. VLDB Endow.*, 5(8), 2012.

Stratis D. Viglas is a Reader in the School of Informatics at the University of Edinburgh. He received a PhD in Computer Science from the University of Wisconsin—Madison and BSc and MSc degrees from the Department of Informatics at the University of Athens, Greece.