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

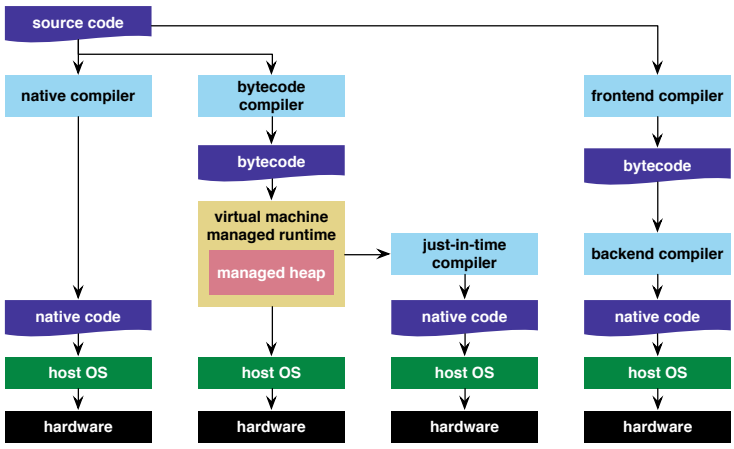

Figure 1: The options when compiling and executing source code: from source code to native code; from source code to bytecode to be executed in a managed runtime, either in an interpreted or in a just-in-time compiled way; from source code to bytecode for a virtual machine to be recompiled by a retargetable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

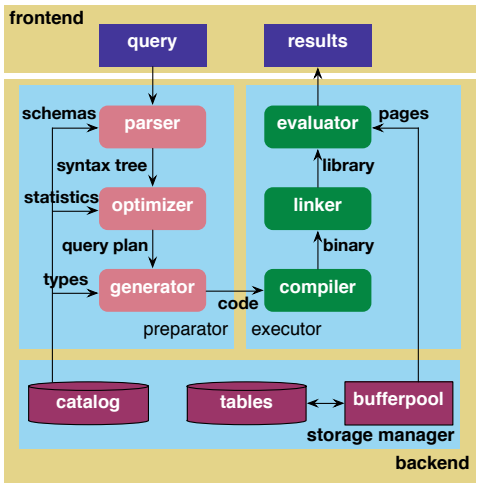

Figure 2: The architecture of HIQUE, the Holistic Integrated QUery Engine [25]

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; }}
```

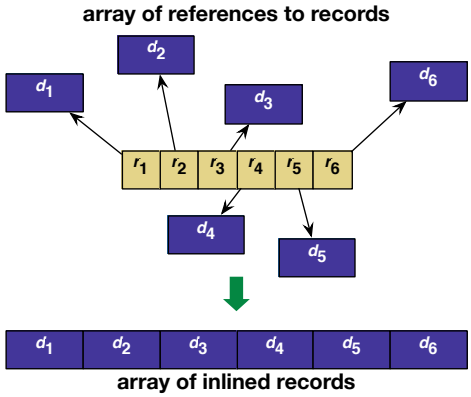

Figure 3: Standard *vs.* inlined representation of tabular data; the second option is closer to an IMDB memory layout.

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.