Processing declarative queries through generating imperative code in managed runtimes

Stratis D. Viglas
Google, Inc. & University of Edinburgh, UK
sviglas@google.com

Abstract

We present the results of our work on integrating database and programming language runtimes through code generation and extensive just-in-time adaptation. Our techniques deliver significant performance improvements over non-integrated solutions. Our work makes important first steps towards a future where data processing applications will commonly run on machines that can store their datasets entirely in persistent memory, and will be written in a single programming language employing higher-level APIs and language-integrated query.

1. Introduction

The falling price of main memory has led to the development and growth of in-memory databases. Whereas new advances in memory technology, like persistent memory, make it possible to have a truly universal storage model, accessed directly through the programming language in the context of a fully managed runtime. This environment is further enhanced by language-integrated query, which has picked up significant traction and has emerged as a generic, safe method of combining programming languages with databases for considerable 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. Our proposal is to apply just-in-time code generation and compilation techniques that have recently been developed for general query processing. The idea is that instead of compiling queries to query plans, which are then interpreted, the system generates customized native code that is then compiled and executed by the query engine. At the same time, we must enable the runtime to take advantage of advances in main memory technology and, primarily, persistent memory. Persistent memory is byte-addressable, but exhibits asymmetric I/O: writes are typically one order of magnitude more expensive than reads. Byte addressability combined with I/O asymmetry render the performance profile of persistent memory unique. Thus, it becomes imperative to find new ways to seamlessly incorporate it into data processing in managed runtimes. We do so in the context of fundamental query processing operations and introduce the notion of write-limited algorithms that effectively minimize the I/O cost. We give a high-level API that enables the system to dynamically optimize the workflow of the algorithms; or, alternatively, allows the developer to tune the write profile of the algorithms. This dynamic adaptation fits in well with the notion of just-in-time compilation.

2. Language-integrated query

Consider the architecture of a typical multi-tier application. The developer primarily decides on application logic: the data structures and algorithms to implement the use-cases of the automated process. The data persistence layers of the application are typically offloaded to a relational database system. Historically, the database system has been optimized for secondary storage. It is accessed through its own query language (typically SQL) through bindings from the host programming language. The developer therefore has to operate over 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 typically 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 environment. Just-in-time query compilation into native code aims to resolve this suboptimality.

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, while, at the same time, borrowing ideas from compiler technology. Our stance is to internally bridge the two runtimes as much as possible so host-language information is propagated to the query executor; and database-friendly memory layouts and algorithms are used to implement the querying functionality [2], [3], [4], [6].

3. Data processing in persistent memory

When memory becomes persistent, the first step in enabling data processing is ensuring that persistence is readily accessible to the higher level programming substrates. This means that the data structures chosen by the programmer are persistent and recoverable. We have designed and implemented a recovery substrate for imperative languages termed REWIND, which stands for *REcovery Write-ahead system for In-memory Non-volatile Data structures* [1].

Our processing model is that data is on byte-addressable persistent memory, accessible directly from user code through CPU loads and stores. Traditionally, data updates are first performed in volatile memory. It is thus possible to delay making log entries persistent until the transaction commits or the data updates are purged from main memory. In REWIND, updates are done directly on persistent memory data: the log entries must be made persistent immediately, and ahead of the data updates. We achieve this through enhanced versions of memory fences (*i.e.*, barriers that enforce ordering and persistence to preceding instructions), cacheline flushes and non-temporal stores (*i.e.*, direct to persistent memory stores that bypass the cache) with persistence guarantees.

Assuming strong software support for language-integrated query and data collections, the next step is dealing with the persistence aspects of the new environment, namely data processing algorithms. We have focussed on two types of data-centric operations: sorting and hash-based relational join processing. Sorting is ubiquitous in a host of data processing algorithms and solutions. Whereas hash-based relational join processing builds on the powerful technique of splitting a dataset in disjoint partitions. Sorting and hash partitioning are used in data mining, (*e.g.*, producing association rules), machine learning (*e.g.*, clustering), and graph management (*e.g.*, nearest neighbor search).

We have devised a family of algorithms that we term *write-limited* that focus on mitigating the write cost of persistent memory for sorting and hash-partitioning [5]. The algorithms are based on a simple observation. Consider the simple process of reading an input dataset and then writing it—perhaps by applying a total ordering on it, as is the case of sorting; or by applying a hash function, as is the case for partitioning. Assume now a write-to-read cost of λ, meaning that writing is λ times more expensive than reading. Then for the cost of writing the output, we can afford λ extra reads. We therefore leverage this ratio to trade writes for reads. The result is that we can achieve the same I/O performance as well-known algorithms (*e.g.*, external merge-sort) but at a fraction of its write cost. Or, alternatively, the developer can tune the write intensity of the algorithms for a small hit on performance. With the algorithms in place, we propose a flexible API that records a blueprint of each algorithm's computation and enables the system to dynamically decide whether to trade writes for reads. The key notion is that the generation of new datasets (be they results of computation, or intermediate structures) is deferred by default. The system keeps track of the accumulated savings and the potential cost associated with generating a dataset. It then performs a dynamic cost-benefit analysis to decide if materializing the dataset would be more cost-effective than deferring its materialization.

References

[1] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(1), 2015.

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

[3] F. Nagel, G. Bierman, A. Dragojevic, and S. D. Viglas. Self-managed collections: Off-heap memory management for scalable query-dominated collections. In *EDBT*, 2017.

[4] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. *PVLDB*, 7(12), 2014.

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

[6] S. D. Viglas, G. M. Bierman, and F. Nagel. Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Eng. Bull.*, 37(1), 2014.