Reifying and Optimizing Collection Queries for Modularity

Paolo G. Giarrusso
Klaus Ostermann
Philipps University Marburg

Michael Eichberg
Software Technology Group,
Darmstadt University of Technology

Tillmann Rendel
Christian Kästner
Philipps University Marburg

Abstract

Conventional collection libraries do not perform automatic collection-specific optimizations. Instead, performance-critical code using collections must be hand-optimized, leading to non-modular, brittle, and redundant code.

We propose SQuOpt, the Scala Query Optimizer, a *deep embedding* of the Scala collection library performing collection-specific optimizations automatically without external tools or compiler extensions.

Categories and Subject Descriptors H.2.3 [*Database Management*]: Languages—Query languages; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.5 [*Programming Techniques*]: Object-oriented Programming

Keywords Deep embedding; query languages; optimization; modularity

Introduction

In-memory collections of data often need efficient processing. For on-disk data, efficient processing is already provided by database management systems (DBMS), thanks to their query optimizers which support many optimizations specific to the domain of collections. However, moving in-memory data to DBMSs does not typically improve performance [6], and query optimizers cannot be reused separately, since DBMSs are typically monolithic and their optimizers are deeply integrated. A few collection-specific optimizations, such as shortcut fusion [1], are supported by compilers for purely functional languages such as Haskell, but the implementation techniques do not generalize to many other optimizations, such as support for indexes. In general, collection-specific optimizations are not supported by the general-purpose optimizers used by high-level (JIT) compilers. Therefore, when collection-related optimizations are needed, programmers perform them by hand.

Some optimizations are not hard to apply manually, but in many cases become applicable only after manual inlining [4]. But manual inlining modifies source code by combining distinct functions together, while often distinct functions should remain distinct to preserve modularity, for instance to separate different concerns or to allow reusing a code fragment. In this case, manual inlining will reduce modularity.

To sum up, developers have to choose between modularity and performance when writing queries. We propose instead an automatic optimizer supporting both inlining and collection-specific optimizations, combining performance and modularity. In particular, our optimizer automatically performs various collection-specific algebraic optimizations, such as map fusion, selection pushdown, automatic indexing and query unnesting. Some of these optimizations can reduce the complexity class of a query. Moreover, the optimizer supports general-purpose optimizations, such as inlining and various algebraic simplifications, to allow collection-specific optimizations to trigger more often. We show a few examples in the next subsection.

Motivating Example

Let us consider for instance map fusion, which combines multiple map operations to avoid intermediate results.

Consider this Scala function definition in module `M1`:

```
def firstPart(someColl: List[Int]) = someColl.map(x ⇒ x + 1)
```

This code defines function `firstPart` which maps function $\lambda x.x + 1$ on some collection of integers `someColl`. Suppose now that module `M2` contains:

```
val theColl: List[Int] = ...
def secondPart = M1.firstPart(theColl).map(x ⇒ x + 2)
```

This code defines function `secondPart` which maps function $\lambda x.x + 2$ on the result of `M1.firstPart`. We assume that these functions are part of different modules, for instance because they are related with different concerns and `firstPart`'s implementation should be hidden behind an abstraction barrier.

This code is inefficient: Executing this code will build a collection to represent the result of `firstPart`, and then consume it immediately to build a new collection; this interme-

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

diate step is unnecessary and expensive. Moreover, we sum each number with first 1 and then 2; adding 3 directly would be faster.

To improve performance we can write, in module `M2`, just `def secondPart = theColl.map(x ⇒ x + 3)`. However, this code combines code fragments which belong to different modules, which is undesirable in our example. An alternative would be to rely on an automatic optimizer. Automating this optimization requires inlining the call to `firstPart` (which in general might be a virtual call, hence resolved only at run-time) and then performing collection-specific optimizations (here, map fusion). While most general-purpose automatic optimizers can handle inlining, they cannot handle collection-specific optimizations. Furthermore, virtual calls typically cannot be resolved statically, hence can only be inlined at run-time by JIT compilers. Hence, this optimization cannot be performed automatically by a typical optimizer, especially not by a compile-time one.

There are many other optimizations that are possible to improve performance, but that require implementation overhead or lead to nonmodular solutions; one example is maintaining indexes to speed up queries. Consider an address book application, which manipulates a collection of people. To find people by their name, this application will probably maintain an index mapping names to people. Each time a person is added or removed, both the collection of people and the index need to be updated: Keeping in sync these operations by hand is error-prone. Automatic index maintenance, as done by databases, would make such inconsistencies impossible, once again reconciling modularity and performance.

Our Solution

To provide automatic collection-specific optimizations and thus address these problems, we introduce SQuOpt, the Scala Query Optimizer, which consists of:

1. An embedded domain-specific language (EDSL) for queries on collections. This EDSL corresponds to the purely functional portion of the Scala collection API, inherits its advantages [3] and naturally supports advanced features of database query languages.
2. A Scala implementation of the above EDSL. Queries written in this EDSL produce explicit representations of themselves, termed expression trees. Crucially, this representation includes Scala expressions and arguments of query operators like `map` or `filter`. This library requires no compiler extensions, unlike LINQ. To this end, we extend existing techniques based on Scala implicit conversions [5] to cope with operations on collections.
3. A run-time compiler to convert queries to runnable code.
4. A run-time optimizer which transforms queries into faster ones before compilation. Since optimizations happens at run-time, handling virtual calls precisely is possible. Since the query representation represents faithfully the arguments of query operators, each optimization has all the information needed to verify that its side conditions apply. While the optimizer is not yet complete, the speedups it achieves are already promising, as detailed subsequently.
5. A prototype implementation of incremental view maintenance, which will allow to materialize and update the results of any view, including indexes, so that the materialized results can be reused in other queries.

Our current implementation is available online.[1]

Overall, our approach seems a promising solution to combine performance and modularity.

Initial experiments

To evaluate our optimizer, we reimplemented some queries (sampled from FindBugs [2]) in Scala. At run-time, we create indexes and measure the optimization speedup on different queries, compared with native queries in Scala. Around half of the queries can be optimized, and speedups range from 2.5x to 2274.9x. In other cases, the optimizer does not improve query performance significantly. Moreover, we study the overhead introduced by dividing queries into smaller functions to increase reuse; our optimizer prototype seems to already reduce this overhead significantly.

Acknowledgements The authors would like to thank Sebastian Erdweg and the OOPSLA anonymous reviewers for their helpful comments. This work is supported in part by the European Research Council, grant #203099.

References

[1] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232. ACM, 1993.

[2] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[3] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *IARCS Conf. Foundations of Software Technology and Theor. Comp. Science*, volume 4, pages 427–451, 2009.

[4] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12(4-5):393–434, 2002.

[5] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010.

[6] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Int'l Conf. Very Large Data Bases*, pages 1150–1160. VLDB Endowment, 2007.

[1] `http://www.informatik.uni-marburg.de/~pgiarrusso/SQuOpt`