Project 2 Specification
Compilation Techniques for Database Systems

April 19, 2016

1 Introduction

This project is mainly about giving you a taste and basic understanding of compilation and high-level programming techniques such as multi-stage programming to achieve better query execution performance. The task we ask of you is simple, but requires that you familiarize yourself with and use an existing compilation framework called SC.

1.1 Fundamentals

1.1.1 DSL Design and Implementation

The SC compilation framework focuses on aggressively optimizing programs written in domain-specific languages such as SQL or relational algebra, where domain-specific optimizations can be applied that general-purpose compilers do not know about. The Development Process[1] document explains the ins and outs of DSL design in Scala, as well as the approach we take.

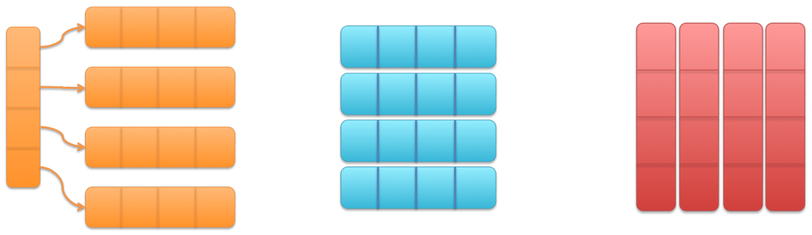
1.1.2 Multi-Stage Programming & Quasiquotation

For an introduction to multi-stage programming and quasiquotation, see the relevant wiki page.[2] If you have problems with the quasiquotes and want to debug your program, also see the wiki.[3]

1.2 Schema Specialization

When the schema of a database does not change often (or never changes), it pays off to specialize query execution code to the current schema of the database, so that for example we statically know the arity of each table, etc. You should read the tutorial hosted on the `sc-public` repository[4] to get a good idea of how this is done in SC.

[1] `https://github.com/epfldata/sc-public/blob/master/doc/DevProcess.md`
[2] `https://github.com/epfldata/sc-public/blob/master/doc/MSP-QQ.md`
[3] `https://github.com/epfldata/sc-public/blob/master/doc/DebuggingQuasiquotes.md`
[4] `https://github.com/epfldata/sc-public/blob/master/relation-dsl/README.md`



(a) Pointer layout (b) Row layout (c) Columnar layout

Figure 1: Different data-layout representations.

1.3 Row and Column Store

As explained in the lecture, there are several possible ways to store the data of a database table. One of the main decisions is whether the memory should be laid out in columns or in rows: either store one single array containing each record – the so-called row store, or a record containing one array for each field f, each such array containing the value of f for all records – the column store. Figure 1 presents a schematic view of these two approaches: (b) is row store and (c) is column store.

In Scala, it is not possible to define C-like structs. One can only define classes (or case classes), which are allocated on the heap. As a result, if we store class instances in an array, we actually store class references (i.e., pointers), and we get the “pointer layout” presented in Figure 1 (a). In this project, we call that the “record store”.

2 Project Description

2.1 Existing Infrastructure

In this project, we provide you with a DSL for relational algebra called `relation`,[5] along with the SC-related infrastructure to deeply-embed it and, in the sbt sub-project `relation-deep`:

- A basic compiler in `RelationCompiler.scala` that applies optimizing transformations and outputs Scala code.
- An abstract class `RelationLowering` that does static schema analysis, extracting static schemas when possible in order to apply specialization.
- A record-store-based specializer `RecordsLowering`, which implements `RelationLowering` and compiles operations of the `relation` DSL into optimized low-level code. Note: This class uses SC records to store data (methods `__new` and `__struct_field`), which automatically generate the

[5] `https://github.com/epfldata/sc-public/tree/master/relation-dsl`

appropriate case classes in the final program. You do not have to use them in your own code.

- A skeleton for the column-store-based specializer you have to write, `ColumnStoreLowering`, which should also implement `RelationLowering`.

2.2 Your Task

Your task is to complete the skeleton in `ColumnStoreLowering.scala` so it compiles operations of the `relation` DSL into optimized low-level code that stores data in columns rather than in rows. You can take inspiration from `RecordsLowering.scala`, which implements the same interface and demonstrate the use of quasiquotes.

More precisely, you have to give a definition for `type LoweredRelation` (replace '`= Nothing`') and implement all method stubs (replace '`???`').

Important: We emphasize that you should implement the `RelationLowering` interface, without changing it. You may define helper methods inside the class, but the signatures of the existing methods should not be changed. The only file you should have to change is `ColumnStoreLowering.scala`.

2.3 Testing Your Code

You can test the `RecordsLowering` and `ColumnStoreLowering` code generators by commenting/uncommenting lines 21 and 22 of `RelationCompiler.scala` and then running the `Main` file of the `relation-deep` project.[6] This can be done with the command `sbt relation-deep/run`. We provide several alternative queries to be selected in the `Main` file. **Make sure your code works for all of them.**

The step above will only *generate* Scala code for a given query. **You also have to make sure the generated code[7] compiles and works**, and that your code for `ColumnStoreLowering` is efficient – it should be comparable in speed to the `RecordsLowering`. Testing the generated code can be done with the command `sbt relation-gen/run`.

2.4 Grading

To grade your solution, we will test it against several queries and datasets, evaluating the correctness of the generated code. Your solution must implement the column-store layout. If it does not, you will not get any points. We will also make sure it is sufficiently efficient, in the sense that it should not be significantly slower than `RecordsLowering` for the queries we gave you.

[6] Located in `/sc-public/relation-dsl/relation-deep/src/main/scala/relation/compiler/Main.scala`

[7] By default, code is generated in `/sc-public/relation-dsl/generator-out/src/main/scala/GenApp.scala`

Important: *In order to be graded, your solution should be able to compile and run from the* `relation` *project's root folder, with the command* `sbt relation-deep/run relation-gen/run`*. If the sbt subproject* `relation-deep` *does not compile, you will be awarded the grade of* ***0****. If the code generated in* `relation-gen` *does not compile, we will have to look at the code manually and your grade will be deducted by a significant amount.*

2.5 Deliverable

The deliverable consists in the single file `ColumnStoreLowering.scala`,[8] that you will have completed. We should be able to replace the provided skeleton with your file and compile without problems.

References

[1] A. Shaikhha, I. Klonatos, L. E. V. Parreaux, L. Brown, M. Dashti Rahmat Abadi, and C. Koch. How to Architect a Query Compiler. In *SIGMOD 2016*, 2016.

[8]Found in `/sc-public/relation-dsl/relation-deep/src/main/scala/relation/compiler/`