A programming language & compiler view on Data Management and Machine Learning systems

Lightweight Modular Staging

is a multi-stage programming approach to facilitate dynamic code generation. The goal is to create efficient low-level code from high-level code. This is achieved, among other things, by replacing certain data types with wrapper types which instead of executing operations just emit source code for these operations. The resulting code is just the actual execution plan without the overhead of the abstractions of the original code.

Futamura Projection

is a special case of partial evaluation. Partial evaluation describes the process where the input data a of program known at compile time gets baked into the program to create a new program that can process the input unknown at compile time faster. An interpreter can be seen as a program that takes a source program and the input and calculates the result. The first Futamura projection is the specialisation of the interpreter for any given source code. The result is a compiled version of that source code.

An Improved Query Compiler

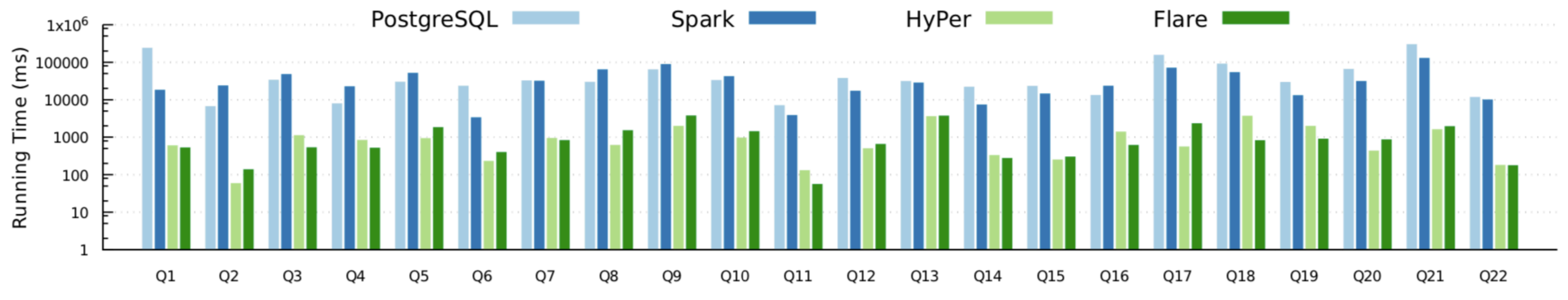
Using the ideas described above a significant speedup for query execution can be achieved. Normally a query gets transformed to a logical and then physical execution plan. This gets executed by an interpreter. By using Lightweight Modular Staging the interpreter only generates source code which will execute the query. This is the first Futamura Projection. The resulting program is a compiled version of the original query without the overhead of the interpreter architecture and higher order data structures used. This overhead can take up more than 80% of the execution time allowing for significant speed ups. Other approaches to compile the database queries based on the llvm interface provide comparable speedups but require specialised code to be written for that purpose instead of the drop-in oriented design of Lightweight Modular Staging.

PostgreSql, Spark - interpreter based implementations

HyPer - llvm based implementation

Flare - LMS based variant of Spark

Speedups of up to two orders of magnitude are possible in a single core environment.



Continuations

Continuations encapsulate the ability to resume the program at a given point in the execution process. They can be implicitly or explicitly passed to a function which can then resume them whenever necessary. This enables, among other things, passing the 'rest' of a program as an implicit parameter. The called function can then execute their code before and after the continuation gets executed. Since continuations preserve the stack but not the values of the variables on the stack the called function can access the potentially changed values after the continuation has been executed.

Applications in Machine Learning

When continuations are combined with LMS it is possible to generate code both for the forward and backward propagation passes. The value gets updated before the continuation is called. This constitutes the forward pass. Then after the forward pass is done the code after the continuations is called. Since this gets called in reverse order the back propagation algorithm can be used to calculate the derivative step by step for the entire formula. This approach has the benefit of enabling any form of function to be learned using backpropagation. Limiting oneself to rigid forms like layered matrix multiplication isn’t necessary. Generating code this way requires again only minimal specialised code as only the methods of the wrapper classes need to implement their own derivative calculations. When continuations can be passed implicitly only the forward pass needs to be written explicitly.

Mino Böckmann
BA IT-Systems Engineering

Hasso Plattner Institute, Potsdam, Germany

E-Mail: mino.boeckmann@student.hpi.de