Database Query Optimization Using Compilation Techniques

Mohammad Dashti Rahmat Abadi
DATA, I&C, EPFL

***Abstract*—Program optimization is an open issue with plenty of methods applied in this area, which is done either manually (by programmers) or automatically (by compilers and optimization modules). This issue gets bolder for DSL optimization, because usually more domain-specific optimizations are applicable, and a higher performance is expected.**

Among all DSLs, we are targeting the ones for querying a database, e.g. SQL. Extensive use of compiler optimization techniques for executing programs in these DSLs can lead to an improved performance in query evaluation. In this paper, we will present the ideas and some techniques used in this direction and their impacts on query execution performance.

***Index Terms*—Compiler Optimization, Staging, Database Query, Compilation Technique**

I. Introduction

As main memory grows, raw CPU cost of processing a query is becoming the means for query performance measurement, rather than traditional cost models that were solely based on I/O costs [1]. This change in cost models should be reflected in DBMS[1]s by utilizing methods that tend to decrease CPU cycles needed for executing a query.

Traditionally, databases use a DSL[2] for communicating with the outside world, and among these languages, SQL is the de-facto standard for this purpose in the software industry. These DSLs have a limited power compared to general purpose imperative programming languages, but these are easier to use and complete enough for the most data definition and manipulation targets.

Moreover, databases use the features of these languages to find the best possible method for information storage and retrieval. A given SQL query is translated into an expression in (physical) algebra, and then evaluation of this algebraic expression will produce the query result [1]. There is a sound literature behind the former task for generating an efficient execution plan, but the latter one has received less attention because it seems like a trivial step that executes the generated plan, using an interpreter.

However, targeting CPU costs, an efficient execution of a given algebraic expression becomes an interesting issue, because it will have a direct impact on overall query execution costs. There are several techniques that can have a positive effect in this part, ranging from optimization techniques used in compilers of general programming languages, to other approaches like *frequency reduction* and *pre-computation*. These refer to computations that can be moved to contexts in which they will be executed less frequently, or to places that efficiency is less critical. These methods are called *staging transformations* in the literature.

Using a powerful and extensible staging framework enables us to define general optimization techniques, as well as other possible optimizations for a DSL, is one possible approach that we have already used and obtained impressive performance results by utilizing it.

The aim of this research is to improve upon the state of the art by providing a method for consistent usage of compilation techniques in efficient execution of database queries, while using intuitions from both database and compiler literature. Dealing with current shortcomings in efficient execution of query plans, or possible changes in prior steps that can lead to an execution plan for a query that is more efficient in the matter of CPU costs, is one of the top goals in this research.

The paper is organized as follows. First we introduce the background by presenting the ideas and techniques used in related works. Then, the current achievements in extending the previous ideas are presented; and, finally we summarize our findings and propose new directions for research.

II. Background

In this section, we will do a literature review about using compilation techniques in the database world and related

Proposal submitted to committee: September 3rd, 2013; Candidacy exam date: September 10th, 2013; Candidacy exam committee: Prof. Karl Aberer (exam president), Prof. Christoph Koch (thesis director), Prof. Viktor Kuncak (co-examiner).

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(M. Dashti Rahmat Abadi) (signature)

Thesis director: _____
(C. Koch) (signature)

Doct. prog. director: _____
(B. Falsafi) (signature)

EDIC-ru/05.05.2009

[1]Database Management System
[2]Domain Specific Language

compiler optimizations that can improve this combination. Here, we will mostly focus on works done by Neumann [1], Rompf et al. [2], and Coutts et al. [3].

A. Query Processing Architecture

Most database systems translate a query into an expression in a (physical) algebra, and then evaluate this expression for producing query results [1]. This evaluation is traditionally done using an iterator model, which is also known as Volcano Style Processing [4]. In this model, each operator receives a tuple stream from each of its inputs, and for iterating over one of them, it should only call *next* operator of the stream to obtain the successor element.

Iterator model, despite its simplicity, flexibility and popularity, is not suitable for main memory databases [1], because: First, it is required to call *next* for every single tuple. Second, invoking *next* is usually a call via a function pointer or a virtual call, which compared to a regular invocation is much more expensive and also decreases the precision of branch prediction in modern CPUs. Third, this model often results in a code with bad locality and frequent instruction mis-prediction, and requires complex state-memorization for storing the current state of execution.

Modern systems change this model by introducing block processing features for receiving more than one tuple in each call to *next*. Amortizing the cost of calling another operator as well as ability to perform vectorized operations can be listed as advantages of this method, but its worst disadvantage is lack of ability to pipeline in this approach. Pipelining means that an operator can pass data to its parent operator without copying or otherwise materializing the data [1].

Although algebraic operator model is good for reasoning over the query and applying several optimizations, it might not be suitable for query processing and producing final results. For more efficient query evaluation, another novel viewpoint is using query compilation strategy [1]. In this approach, algebraic operator model is still used for the first part related to reasoning about query, but it differs from that model: First, it is possible to do data-centric query processing, rather than an operator-centric approach. Second, data is pushed toward operators, except being pulled from operators. Third, queries are compiled and optimized further into native machine code, rather than being interpreted each time.

Moreover, using compilation strategy for query processing will produce executables that rival hand-coded query execution plans and even in some cases, outperform hand-written code. In addition, we automatically benefit from future compiler optimizations and hardware optimizations, almost for free [1].

A high level architecture for query processing is shown in Fig. 1. This architecture is conceptually derived from [1] and its main purpose is to reflect his idea about the best place for putting the query compiler module. It argues that instead of directly compiling a SQL query, we should leverage the benefits of possible optimizations in algebraic model and the output of this step can be used as input of query compiler module, which will generate the final query evaluator engine.

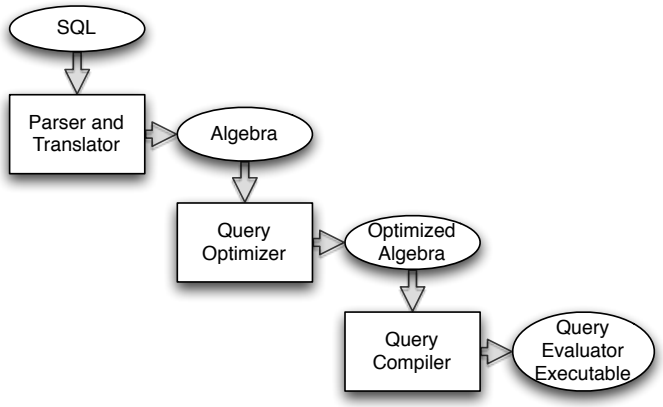

Fig. 1: A high-level query processing architecture using compilation strategy (proposed in [1])

B. Query Compilation

There are several previous efforts targeting query compilation. In [5] the authors proposed a method for compiling query logic into Java Bytecode which allows for using the JVM[3]. However, this is relatively heavy weight [1] and they still use iterator model in their approach which limits the benefits.

Another work on HIQUE system [6] suggested the query compilation into C code using code templates for operators. By inlining the result of materialization inside the operator execution, HIQUE removes the iterator model. However, compared to approach used in [1], operator boundaries are still visible and there is a high cost of compiling generated C code.

In [1] the author proposes a new concept, named *pipeline-breaker*. A pipeline-breaker is an algebraic operator that for a given input side, takes an incoming tuple out of CPU registers. The author himself admits the definition is slightly hand-waving, because a single tuple might already be larger than available CPU registers, but he assumes that we have enough registers. The main point about a pipeline-breaking operation is the fact that we need to spill data into memory, when we want to apply this operation.

Moreover, it is proposed in [1] to reverse the direction of data flow control, so data stream is pushed to operators, rather than being pulled, as it was the case in traditional model. With this assumption, we should continue pushing data until we reach a pipeline-breaker. As a result, data is always pushed from one pipeline-breaker to another pipeline-breaker. There are two advantages in this method: First, enclosed operators do not manipulate tuples in CPU registers, so they are very cheap operations. Second, complicated control flow logic will remain outside of tight loops and will reduce register pressure.

In addition, instead of simple common interface in iterator model that each operator should only implement *next()* function, [1] introduces a new interface consisting of two functions, *produce()* and *consume(attributes, source)*. Such an abstraction helps to keep the code maintainable, even though in this case, it only exists in query compiler and is only a mental model. A sample translation scheme to illustrate the produce/consume

[3]Java Virtual Machine

interaction for generating pseudo code is shown in Fig. 2. Actually, the real translation code is more complex, because it should keep track of the loaded attributes, operators involved in the evaluation, and attribute dependencies if an attribute is referenced by a subquery.

$\bowtie$.produce	$\bowtie$.left.produce; $\bowtie$.right.produce;

$\bowtie$.consume(a,s)	if (s==$\bowtie$.left)
	print "materialize tuple in hashtable";
	else
	print "for each match in hashtable["
	+a.joinattr+"]";
	$\bowtie$.parent.consume(a+new attributes)

σ.produce	σ.input.produce
σ.consume(a,s)	print "if "+σ.condition; σ.parent.consume(attr,σ)
scan.produce	print "for each tuple in relation" scan.parent.consume(attributes,scan)

Fig. 2: A sample translation scheme for showing produce/consume interaction (proposed in [1])

The target language is another considered issue in [1]. There is a possibility to generate code in a higher level language like C++, which can easily connect to other existing systems, but in this case we will have no total control over generated code (e.g for accessing overflow flags). There is also a big discussion in [1] about the compilation time needed for an optimizing C++ compiler to produce an executable, which is a relevant feature in this context, but it might not be a critical feature in some applications that involve using a query for a long term period.

On the other hand, they neither propose generating the whole query processing logic in a low level language, like LLVM[4]. Even though, generating LLVM assembly will give us the opportunity to execute the resulting code using an optimizing JIT compiler provided by LLVM, but writing assembler code is more difficult, and we might have existing database logic implementation in another high-level language like C++ (e.g index structures). Beside that, in some cases it is not even possible (or desirable) to compile a complex query into a single function, because it might lead to exponential growth in code size [1].

All in all, author of [1] proposes a mixed compilation and execution model, which offers implementing complex parts, involving complex data structure management or spilling to disk in C++ and they can be pre-compiled. Then, the dynamically generated code in LLVM will connect different operators. This scheme will result in a smaller generated code with a very fast compilation. However, for optimal performance, care should be taken to preserve the hot path in LLVM, because while we stay in LLVM, we can keep tuples in CPU registers, and calling an external function is expensive, as all registers should be spilled into memory.

C. Program Transformation via Staging

Program transformations can be categorized into *optimizations* and *lowerings*. An optimization converts a given code to a more efficient code in the same level of abstraction, while a lowering only translates a program into a lower level representation. Optimizations do not have any clear ordering, and hence are prone to phase ordering problems, so they should be combined for maximum effectiveness, while lowerings have a clear ordering and undoubtedly should be applied after optimizations, in order to avoid missing high-level optimization opportunities [2].

As a logical result, the direct lowering transformation applied in [1] for generating LLVM or even C++ code and relying on the compiler for optimizations, might not produce the most efficient program, because it might miss some optimizations in the higher level of abstraction.

For addressing these short-comings in program transformation and compilation, extensible compilers were proposed. The goal of these compilers is to reason about user/library defined programs and data structures. The common mechanism used in them is: First, front-end macros, staging or partial evaluation, which all intend to programmatically remove abstractions before they enter compiler, but might not be able to produce the most optimized code, due to no control over what finally happens in the compiler. Second, extending internal workings of compiler by adding new transformation passes at different points in the compile chain (that leads to phase ordering problems) and new IR types (which often it is not clear, how they will interact with existing generic optimizations) [2].

Many computations can be separated into stages based on frequency of execution, or availability of information. Multi-stage programming (MSP, staging for short) as proposed by Taha and Sheard [7] make the stages explicit and makes it possible for programmers to delay evaluation of a program expression to a subsequent stage (thus, staging an expression).

MSP is tightly related to partial evaluation [8], which uses statically known parts of input data in a program for applying specializations. There is an important difference between these two concepts in which partial evaluation strictly specializes programs and usually comes with soundness guarantees whereas adding staging annotations to a program in an MSP language such as MetaOCaml [9] is simpler for composing staged functions but needs special attention for preventing a change in the results of computation [2].

Staging is usually used as a method for program generation. Executing a multi-stage program will produce an object program that can be normally compiled and executed against real input data. Using this technique is known to simplify the process of generating programs, and with the same approach, it can be used for simplifying program transformation. It is straightforward to split any transformation into traversal and generation parts, which staging helps with the generation part, for example using LMS that is a single-staging extension for Scala [2].

Moreover, LMS goes one step further by implementing internal compiler passes as IR interpreters, which are staged themselves and produce a new program as their output. Consequently, they can delegate back to program execution for performing the program transformation [2].

Enumerating advantages of using staging for program transformation, we can mention: First, implementing a (staged) IR

[4]Low Level Virtual Machine

interpreter is much simpler than creating a IR to IR transformer (the way optimization are usually implemented inside compilers). Second, optimizations can be added progressively to a program that uses staging. Third, the staged program (or library) can also control the translation and can affect the generated code [2].

D. Applying Optimization Techniques

There is no optimization stage after the translation phase in [1], and it is assumed the result of translation will be further optimized via target compiler or execution environment. Trying to perform a translation which is easier to optimize in pre-execution steps, is still considered in their work and had a noticeable impact on the evaluations. For example, *loop inversion optimization* (explained later in part 6 of this section) is used and explained in their code generation without explicitly naming it. However, there is still the possibility of loosing some optimizations, due to the lowering which happens in the translation phase [2].

LMS is a library-only staging approach. In contrast with dedicated MSP languages that are based on quasi quotation, the only means of computational stage distinction in LMS is using types. Rep[T] type in the first stage is used for a computation that should yield a result of type T in the next stage, and similarly, T type is used in the first stage for computations that should yield constant result in the second stage. The normal Scala type system gathers and propagates information about staged expressions and therefor executes a semi-automatic local BTA[5]. Now, if type T is already supported by LMS (and you are only using Scala's standard library) you are done and LMS will easily generate code for the next stage, given your program [2].

Then, upon execution of your first stage program including LMS library, LMS does not directly produce the next stage program in a source code format, but instead, as an intermediate representation (IR), which is actually a "sea of node" dependency graph [2]. Each IR node in this graph is either a constant (Const) or a symbol (Sym) which contains an operation. The operation in each Sym might contain other Const, Sym, or Blocks (which is a list of statements, each containing a Sym, preserving the ordinal sequence of operations in that block) [2].

Having this IR graph in hand, LMS can apply different transformations by traversing the graph. Sequences of "optimization, lowering, optimization" should be applied on the IR graph, until we reach the lowest representation. Finally, code generation is just done by a traversal on this graph and a function call for code generation on each node [2].

User programs build their own abstractions and data structure hierarchies as extensions to the language. In these cases, one can easily use high extensibility and modularity of LMS to extend it and make it work for its own data-structures. IR graph formation and code generation are both modular, meaning that you can add required functions for creating IR nodes of your abstractions, as well as functions for generating appropriate code for them. Encapsulation of these extensions in a *trait* makes it re-usable for further usages [2]. We name these extensions as *Lifters*, because they make it possible to lift your program structure to be a part of IR graph.

In addition, LMS provides the ability for manual staging in which users can program functions of type $Rep[A] \Rightarrow Rep[B]$. One important detail here is the difference between types $Rep[A \Rightarrow B]$ and $Rep[A] \Rightarrow Rep[B]$. The former is a staged function object, and macro systems only allow this kind of functions that only allow lifting expression trees and as a result, limits expressiveness, and there is no guarantee that higher order functions are evaluated at staging time. In contrast, the latter is a function on staged values, which allows the function parameter to be evaluated and unfolded at staging time, creating greater opportunities of optimization [2].

Compiler optimizations can be categorized into two types: First, *local optimizations* that aim at optimizing a basic block, using only local information in that block. As there is no control flow structure inside a basic block, analysis cost for these optimizations is very cheap (saving time and reducing storage requirements for applying this type of optimizations) [10]. Second, *global optimizations*, which are also called "intraprocedural methods" and try to analyze and optimize whole functions or program blocks. This gives them more information to operate, but it often requires expensive calculations. Pessimistic assumptions should be made when there are function calls or accesses to global variables (because there is a few information available about them) [10].

Constant folding and specialization as optimizations, have been already mentioned in this section, and are automatically covered by LMS via having different types in each stage. In the following sub-sections, we will introduce other important compiler optimizations and the way each of them were addressed by previous works.

1) Common Sub-expression Elimination (CSE): This optimization intends to prevent redundant computations by performing them only once and re-using the result, whenever it is needed in the posterior computations. LMS applies this optimization by performing an on-the-fly ANF[6] conversion.

In this process, every new expression created in ANF is assigned to an identification symbol and registered in a map collection, and the symbol is returned to be used instead of that expression, in the rest of execution. For every successor expression, it will be first checked whether it exists among previously registered expressions, which in this case, except creating a new symbol, the symbol of existing expression will be returned [2].

2) Dead Code Elimination (DCE): In a computer program, there might exist computations that can be removed, either because they do not contribute to the final result, or the control flow does not allow them to be executed. Eliminating the former case will save a lot of unnecessary computations, and the latter will produce a more compact executable.

As it was mentioned before, every program transformation and code generation in LMS is just a IR graph traversal. Based on this fact, starting traversal from the result symbol, dead nodes are excluded, just by traversing the strongly connected

[5]Binding Time Analysis

[6]A-Normal Form

components of this graph [2].

3) Combining Optimizations (Speculative Rewriting): Among compiler optimizations, *rewriting*s are preferred, because it is easy to specify them separately, and do not require programmers to define abstract interpretation lattices. In LMS, rewrite rules are defined using smart constructors in Lifter classes [2].

Many optimizations are mutually beneficial, and in the presence of loops, optimizations need to make optimistic assumptions for obtaining the best results. If we apply optimizations separately, each optimization should effectively make pessimistic assumptions about the outcome of all others. Combined analyses avoid the phase ordering problem by solving everything at the same time. In LMS, forward optimizations are applied at IR construction time, while loop information is incomplete. In order to avoid changing the semantics of an input program, structured loops are used instead of control flow graphs, and dependency information and rewriting is used instead of explicit data-flow lattices [2].

Commonly, rewriting is semantics preserving, in other words, pessimistic. The idea used in LMS is to drop this assumption. As a consequence, we should rewrite speculatively and be able to rollback to a previous state to get optimistic assumptions. The algorithm works as follows: for each confronted loop, all possible optimizations should be applied to loop body, without any initial assumptions. Then, by analyzing the result of the transformation, if any new information is discovered that needs new assumptions in previous steps, the transformed loop body should be discarded and the original loop should be retransformed using updated assumptions. This cycle should be repeated until the analysis result reaches a fixed-point, which is the final acceptable transformation [2].

4) Compound Expressions (Split and Merge): IR graph in LMS contains structured expressions like loops and conditionals, so optimizations should reason about them, too. This is not easily achievable: for example, a simple DCE algorithm will not be able to eliminate only parts of a compound expression. The solution used in LMS is eagerly splitting many kinds of compound statements, assuming optimistically that only parts will be required. Splitting is implemented just as a rewrite rule, and thus integrates well with other unrelated optimizations. Afterwards, essential parts are distinguished by running the normal DCE algorithm, and then the remaining pieces are reconstructed [2].

5) Data Structure Optimizations: High level data structures are the basis of modern programming and at the same time they might prevent compilers from applying all possible optimizations. For example, in object oriented programming, each new instance of a class should be allocated separately, and this problem becomes bolder when allocating an array of objects is needed [2].

As this overhead is non-negligible, a generic struct interface is used in LMS, which is a generic implementation for Product types (with common optimizations). In this interface, struct creation is equivalent to creation of a map that relates field identifiers (static) to values (dynamic) alongside with a tag field, containing information about data representation. Moreover, field access is just looking up desired value directly from the map, assuming the argument is a struct node. Struct abstraction can be extended to cover sum types (unions) and inheritance, using a tagged union approach by adding a class field to each struct [2].

AoS[7] to SoA[8] conversion is another possible data-structure optimization using struct interface, which converts arrays of objects into an object containing several arrays, one for each field. This way, boxing/unboxing in JVM will be avoided, SIMD execution will be possible, and new chances for eliminating unused parts of a data-structure might occur [2].

6) Loop Inversion: This optimization is rather simple, and there is only a transformation from a *while loop* to an *if block* containing a *do-while loop*. This optimization will reduce the number of jumps by two (if loop is executed), improves branch prediction, and enhances instruction pipelining.

It is argued in [1], that branches are cheap, if branch prediction works, which means that a branch is taken nearly never or nearly always. *Loop inversion optimization* helps branch prediction by distinguishing between cases that we will never get into a loop, and cases that will execute several iterations of the loop. Author of [1] reports a 20% improvement upon using this optimization.

7) Loop Fusion and Deforestation: Loop fusion is an optimization technique and loop transformation for improving memory locality by combining multiple loops into a single one. Deforestation is another program transformation for eliminating intermediate data structures for computing the final result of a program. Applying fusion after inlining functions, usually results in deforestation, as merging operations that traverse over elements of a data structure, normally eliminates intermediate results.

Several shortcut fusion systems were proposed in the literature to address these optimizations. These systems usually use List data type as their target type to apply their techniques, as it is the primary data-structure for functional programming. Even, List is used as a control structure in lazy languages (e.g. Haskell). List operations can be classified as producer (e.g. list constructor), transformer (e.g. map), and consumer (e.g. foldl) operations [3].

Build/foldr [11] is one of the most practically successful shortcut fusion systems. There is a single rule in this system to eliminate adjacent occurrences of the list combinators. Weakness of build/foldr is its failure to handle *foldl* (e.g. sum) which consumes a list using an accumulating parameter, and *zips* that consume multiple lists in parallel. The intuition behind build/foldr is viewing lists as sequences represented by data-structures, and fusing functions that work directly on the structure of that data.

Destroy/unfoldr [12] is another shortcut fusion system which fails at fusing functions on nested lists (e.g. concatMap), list comprehensions, and filter-like functions which must be defined recursively.

Stream fusion [13] is an additional fusion system that was unsuccessful in correctly fusing *nested lists* and *zips*. Unlike *build/foldr*, both *stream fusion* and *destroy/unfoldr*

[7]Array of Struct
[8]Struct of Array

convert list functions to work on dual of the list, which is an unfolding or co-structure. However, *stream fusion* uses explicit representation for list co-structure as *Stream* type, rather than an implicit one used in *destroy/unfoldr* [3].

A new extension to *stream fusion* [3] is a new approach based on "equational transformation" which has a wider coverage than previous shortcut fusion systems, and tries to overcome all their limitations (by covering filter, fold, append, concatMap, list comprehensions, and functions on nested lists).

In fact, Stream type used in [3], encapsulates an *unfold* by wrapping the initial state and stepper function inside itself.

$$\begin{aligned}&\textbf{data}\ Stream\ a\ =\ \exists s.Stream(s \to Step\ a\ s)\ s\\&\textbf{data}\ Step\ a\ s\ \ = Done\\&\qquad\qquad\quad \mid Yield\ a\ s\\&\qquad\qquad\quad \mid Skip\ s\end{aligned}$$

Conversion from lists to streams happens using *stream* and *unstream* combinators. Assuming *stream* . *unstream* as the identity on streams, the only specific optimization used for stream fusion is:

$\forall s\ .\ stream(unstream\ s) \mapsto s$

Moreover, having converted the functions over list structures into functions on stream co-structure, the key trick for creating opportunities to optimize away intermediate Step constructors by composed functions on streams is avoiding to implement stream producers recursively, and doing only one operation on a stream in each step (by using *Skip* state).

The main idea in the stream fusion system is transforming list functions to expose their structure, then the actual optimizations (e.g. eliminating intermediate values) will be applied via general purpose compiler optimizations at compile time. Accordingly, having no control over producing the most optimized program, there is a strong dependency between stream fusion system and Haskell compiler. This tight coupling even results in slowdown for almost half of their benchmark, because some optimizations are not still supported by Haskell compiler.

Besides that, there are other deficiencies in stream fusion system: First, for fusing general recursive definitions, writing stream stepper function is not always easy, and at the same time, representation of control-flow as state (for implementing stream version of list functions) makes them appear inside-out and hard to understand. Second, fusing more general algebraic data types, like Tree, using the same approach needs new infrastructure for each new data structure.

LMS introduces a new approach for loop fusion and deforestation. The core loop abstraction in LMS presented in [2] is

$loop(s)\ \overline{x = \mathcal{G}}\ \{\ i\ \Rightarrow\ \overline{E[x \leftarrow f(i)]}\ \}$

where s is the loop size and i the loop variable getting values from $[0, s)$. Multiple results $\overline{x}$ can be computed in a loop, which each of them is bounded to a generator $\mathcal{G}$. Using this formulation, a generator can be a Collect, which creates a flat array-like data-structure, Reduce($\oplus$), which reduces values with an associative operator $\oplus$, or Bucket($\mathcal{G}$) operation, which produces a nested collection by grouping generated values by key and applying $\mathcal{G}$ to those with matching key. Yield statements $x \leftarrow f(i)$ form the loop bodies, which give values to generators (of current loop or an outer loop). These yield statements are placed in an outer context $E[.]$ that might contain other control structures, like loops or conditionals.

Using this model it is possible to represent many common collection operations, and the fact that LMS does the subsequent required optimizations itself, after applying fusion, it is guaranteed that all intended optimizations will be used for generating output code. However, not all of the loop fusion ideas presented in [2] (without being mentioned) are completely implemented in LMS project. There might be some corner cases and details which would be identified only after the uncut implementation [2].

E. Parallelization

Parallelization can be achieved in several different levels, and there are diverse opportunities in each level.

Block-wise processing, despite its disadvantage of creating additional memory accesses, enables inter-tuple parallelism by processing multiple tuples with one instruction. This is achieved by using SIMD[9] instructions in modern CPUs, which will execute the same operation for a block of multiple input data, as long as we can keep the whole block in registers [1].

Modern CPUs are multi-core (because of frequency limit for a single core), so there are other opportunities for parallelization in this case. For leveraging multi-core processing, nearly all database systems provide inter-query parallelism, that is executing each query on a separate core and you will be able to evaluate multiple queries at the same time [1].

In addition, there are new ideas for using multi-core processing to achieve intra-query parallelism, evaluating a single query faster, using the power of multiple cores. This is applicable by partitioning the input of query processing operators, processing each partition independently and finally merging results from all partitions [1].

The author of [1] without going into details, claims that they can support intra-query parallelism with nearly no code changes. Their generated code always operates on fragments of data which are processed in tight loops and the result is materialized into the next pipeline breaker. These fragments of data are usually determined by the storage system, and they could as well, come from a parallelizing decision. Beside that, only some additional logic is required to merge individual results.

Delite [14] is a parallelization framework for DSLs developed on top of LMS. Its goal is to enable the rapid construction of high performance, highly productive DSLs. In addition to LMS capabilities, it generates an execution graph that targets multiple heterogeneous hardware devices. However, Delite is still in alpha, and there is no official release for it.

III. Compilation Techniques in Action

As a starting point for using compilation techniques for extending previous efforts in efficiently evaluating database queries, we started working on extending LMS and integrating it with DBToaster. DBToaster [15] is a system which creates query engines for embedding into other applications that have

[9]Single Instruction, Multiple Data

use-cases for real-time, low-latency processing and monitoring of their input data.

Query engines generated by DBToaster tend to be optimized for long-running queries, where keeping an up-to-date version of query results is required, while input data changes frequently. This task is done by maintaining in-memory materialized views. The input query language for DBToaster is SQL. Given a SQL query, DBToaster generates the query engine code that easily integrates into the target Scala or C++ project [15].

An illustration for high-level architecture of DBToaster is provided in Fig. 3. This system accepts a SQL query as input, first translates the query to M3 language, which is a trigger-based language for maintaining the result and intermediate collections required for quick calculation of the result of the query. Then, the output in M3 language will be translated into another high-level intermediate language, named K3, which is an abstract functional language. The program in K3 will be optimized based on some pre-defined rewrite rules in K3 optimizer module. Finally, query engine code will be generated using optimized K3 code into Scala or C++ languages.

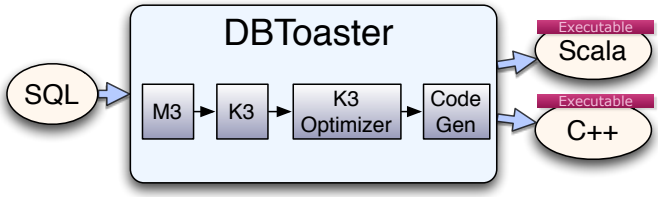

Fig. 3: A high-level architecture of DBToaster

As we discussed about program transformations in this paper, it is clear that using only rewrite rules will not produce the most optimized code. Consequently, we decided to encapsulate all optimizations in a separate optimizer module, named ToasterBooster which is based on LMS project. A schematic of new architecture of DBToaster after integrating ToasterBooster with it, is shown in Fig. 4.

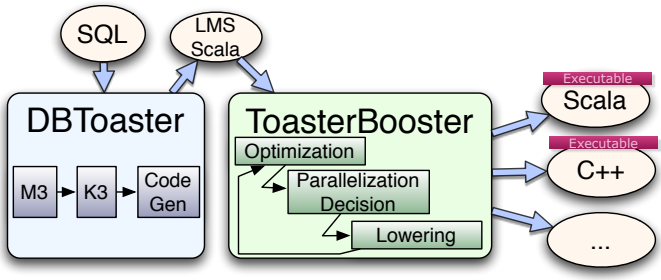

Fig. 4: High-level architecture of DBToaster with ToasterBooster as its separated optimization module

Adding *loop inversion optimization*, a new loop fusion technique, a graphical graph representation generator, and a new smart code generator to LMS are among the most effective enhancements which were added in ToasterBooster project. Beside other general bug fixes applied to LMS core, other extensions were DBToaster specific lifters and their optimizations.

In addition, there are optimizations like loop fusion and deforestation, that are not completely implemented in LMS, and its general implementation is not straight forward. However, optimizing DBToaster did not need general loop fusion support. We only needed to support higher-order functions defined on a collection, like map, flatMap, flatten, foreach, filter, fold, reduce, toList, and etc.

A good idea for dealing with this situation is using $generators$. A generator is a special routine that can be used to control the iteration behavior of a loop. Earlier, Kiselyov et al. [16] proposed a programming style for incremental stream processing based on typed simple generators. We used their idea beside extending it to further stages that are not really straightforward (e.g for $flatten$ method on a collection). Here, we simply define a Generator as a function:

((Rep[T] ⇒ Rep[Unit]) ⇒ Rep[Unit])

As functions are first-class citizens in Scala, we could define them as an abstract class and these can have their own methods and attributes. Here we can define the higher order functions required in our Collection type used in DBToaster, as methods of this class. A sample implementation for a few higher order functions is shown in Figure 5.

```
abstract class Generator[T] extends
    ((Rep[T] ⇒ Rep[Unit]) ⇒ Rep[Unit]) { self ⇒

    def map[T2](g: Rep[T] ⇒ Rep[T2]) =
      new Generator[T2] {
        def apply(f: Rep[T2] ⇒ Rep[Unit]) =
                    self.apply{ x:Rep[T] ⇒ f(g(x)) }
      }

    def foreach(g: Rep[T] ⇒ Rep[Unit]) =
        self.apply{ x:Rep[T] ⇒ g(x) }

    def fold[Y](init: Rep[Y],
      g: Rep[T] ⇒ (Rep[Y] ⇒ Rep[Y])): Rep[Y] = {
        var res = init
        self.apply x:Rep[T] ⇒ res = g(x)(res)
        res
    }
...
}
```

Fig. 5: Using generators as means of deforestation. Generator class definition is defined in this figure. One can use this generator by implementing its only abstract method: apply.

For using this generator, one has to change the interface and core implementation classes. If you have any higher order function in your interface, you should change their return types from e.g. Rep[Collection[T]] to Generator[T]. Then, your core implementation should extend the generator and should only implement the apply method of generator. Depending on the target data-structure, this apply method can be implemented as a $foreach$ method call for a collection data type, or a single function application for single valued types.

Upon execution of the first stage program, seeing that generators work as stream-processors, symbolic execution of the program will lead to a second stage program that contains only low-level operations and $foreach$ method call. Besides that, it will combine operations on same collection type as much as possible.

The main focus of this project was on the optimization of DBToaster's generated programs in Scala, so our main benchmarking target was this system and sample queries that where earlier used in DBToaster publications for showing its performance advantages in contrast to other existing view maintenance systems.

All experiments were performed on a Mac Book Pro 8,2 with one Intel Core i7 2.2 GHz processor and 8GB of RAM. Scala code was run on the Oracle Java SE Runtime Environment 1.6.0_45 and the Hotspot 64-bit server VM with default options. We ran each program six times (to warm up the JIT) and report the average of the last 3 runs. For each run we timed the whole execution of program from reading first input tuple until producing the final output.

The result of these experiments for seven different queries are shown in Figure 6.

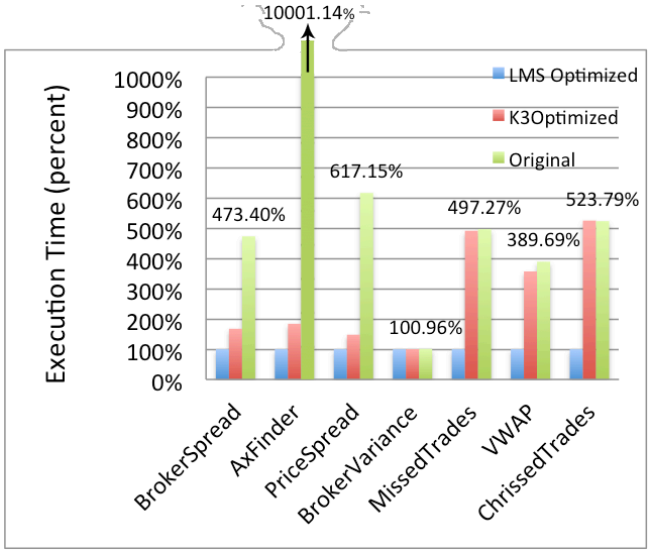

Fig. 6: Performance evaluation of DBToaster's original generated view maintenance program, compared to a previously existing optimizer for that system (named K3Optimizer), and ToasterBooster-optimized version of the original program.

IV. Future Research

Extending the existing methods in the literature and providing new optimization techniques that can have a positive impact on the performance of database query evaluation, and finding the best way for combining them altogether, is the final goal of this research.

Moreover, optimizations work by removing abstractions and intermediate results, which makes it even more difficult to execute the resulting optimized program in a parallel (many cores) or distributed (many machines) settings. "Parallelizing decision" is a difficult problem[1], and hence a potential direction for research in this context.

Another research path is adaptive query compilation. In this approach, instead of deterministic compilation which generates the same resulting code each time, we use a feedback from runtime statistics of program execution, for the next round of compilation. Based on this additional input, compiler might decide to generate a new program as its output, which better fits the runtime workload. This makes it also possible for applying multi-objective query optimization, which different code can be generated at different points in time, to satisfy different constraints.

All in all, with great opportunities in this area of research, we hope that by extending our current works and combining them with some new ideas for adaptive query compilation and parallelization, we will be able to officially publish our robust results in less than a year. After that, we will focus on using compilation techniques on other modules of a DBMS.

References

[1] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, Jun. 2011.

[2] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," *SIGPLAN Not.*, vol. 46, no. 2, pp. 127–136, Oct. 2010.

[3] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: from lists to streams to nothing at all," *SIGPLAN Not.*, vol. 42, no. 9, pp. 315–326, Oct. 2007.

[4] G. Graefe and W. J. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in *Proceedings of the Ninth International Conference on Data Engineering*, 1993, pp. 209–218.

[5] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman, "Compiled query execution engine using jvm," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE '06, 2006, p. 23.

[6] K. Krikellas, S. Viglas, and M. Cintra, "Generating code for holistic query evaluation," in *ICDE*, 2010, pp. 613–624.

[7] W. Taha and T. Sheard, "Metaml and multi-stage programming with explicit annotations," in *Theoretical Computer Science*. ACM Press, 1999, pp. 203–217.

[8] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[9] C. Calcagno, W. Taha, L. Huang, and X. Leroy, "Implementing multi-stage languages using asts, gensym, and reflection," in *Proceedings of the 2nd international conference on Generative programming and component engineering*, ser. GPCE '03, 2003, pp. 57–76.

[10] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[11] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the conference on Functional programming languages and computer architecture*, ser. FPCA '93, 1993, pp. 223–232.

[12] J. Svenningsson, "Shortcut fusion for accumulating parameters & zip-like functions," in *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ser. ICFP '02, 2002, pp. 124–132.

[13] D. Coutts, D. Stewart, and R. Leshchinskiy, "Rewriting haskell strings," in *Proceedings of the 9th international conference on Practical Aspects of Declarative Languages*, ser. PADL'07, 2007, pp. 50–64.

[14] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011, pp. 89–100.

[15] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic, "Dbtoaster: higher-order delta processing for dynamic, frequently fresh views," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 968–979, Jun. 2012.

[16] O. Kiselyov, S. L. P. Jones, and A. Sabry, "Lazy v. yield: Incremental, linear pretty-printing," in *APLAS*, 2012, pp. 190–206.