McGill University
School of Computer Science
Sable Research Group

HorseIR: Fusing Array Programming and Database Query Processing

Sable Technical Report No. McLAB-2018-03

Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme and Laurie Hendren

January 30, 2018

Contents

List of Figures

List of Tables

Abstract

While traditional relational database management systems (RDBMS) seem to be a natural choice for data storage and analysis in the area of Data Science, current systems still fall short in two aspects. First, with increasingly cheap main memory, many workloads now fit into main memory while RDBMS have been optimized for I/O. Second, while current systems support advanced analytics beyond SQL queries through user-defined functions (UDF) written in procedural languages, integration into the SQL engine mostly follows a black-box approach limiting the opportunities for a holistic optimization. In this paper, we propose HorseIR, an array-based intermediate representation that allows for a unified representation of UDFs and SQL execution plans optimized with traditional RDBMS optimization techniques. HorseIR has a high-level design, supports rich types and data structures, including homogeneous vectors and heterogeneous lists. We identify suitable optimizations for generating efficient code from HorseIR, taking memory and CPU aspects into account. We compare HorseIR with the MonetDB RDBMS, by testing both standard SQL queries and queries with UDFs, and show how our holistic approach and compiler optimizations benefit the runtime of complex queries.

HorseIR: Fusing Array Programming and Database Query Processing

Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme and Laurie Hendren

January 30, 2018

1 Introduction

Relational Database Management Systems (RDBMS) have been the primary data management software of choice for organizations for decades with SQL being the de facto standard query language [27]. Being a declarative language based on relational algebra [17], SQL gives RDBMS implementors the opportunity to optimize the execution plan for a query [13]. However, these optimization efforts are primarily targeted towards reducing disk access, as I/O has been the primary resource bottleneck for the most part of RDBMS history [28]. Studies have shown that the increase in the size of main memory is resulting in many RDBMS workloads becoming compute and memory bound as disk blocks are being cached in the memory [7, 11], therefore warranting more sophisticated optimization strategies.

Additionally, the growing popularity of fields such as data science and machine learning has resulted in an increase of advanced analytical applications that employ statistical and learning algorithms. SQL has structural limitations which make it difficult to support many complex linear algebraic and procedural logic required in these applications. Therefore RDBMS vendors are turning to more conventional procedural languages to address this functionality gap by providing the feature of a User Defined Function (UDF) that can be written in a conventional procedural language and invoked as part of a regular SQL query [33]. However, for most part this integration has followed a black box approach where the SQL optimizer has no insight into the logic of UDF and vice versa, thus forfeiting any opportunities for a holistic optimization. Only recent efforts [19, 18] attempt to address this performance problem, e.g., by having UDF analyzers gather statistical information inside compiled UDFs.

From a programming language (PL) point of view, many important data analytics and machine learning algorithms are currently being implemented in array-based programming languages such as Python (NumPy) [8], R [40], and MATLAB [2]. Historically, many data analytic approaches, including many financial computations, were done with APL.

Our approach, and the main objective of this paper, is to tie together the strengths of modern database technology and modern array-based PLs by providing a common, array-based intermediate representation (IR), that we call HorseIR that serves two main functions. (i) By translating SQL execution plans to HorseIR, many standard compiler optimizations and parallelizations can be applied to HorseIR which leads to improved performance for CPU-bound and memory-intensive database queries. (ii) A common array-based IR facilitates optimizations over the PL and SQL-

based boundary, enabling performance improvements that are not possible when the RDBMS and PL are developed as separate entities. Our overall approach is shown in Figure 1, which demonstrates that both the UDFs and the SQL queries are translated to HorseIR, and then optimized together, in order to provide efficient target code.

Research has shown that RDBMS implementations optimized for large scale data processing tend to benefit from a column-store architecture, an approach where each column is stored contiguously on its own [4, 5]. Here a column's logical data structure is akin to that of a programming language array. Therefore, the choice of an array-based IR is very important. Working datasets[1] of column-stores are good candidates to apply array programming optimizations as a column essentially contains homogeneous data which maps nicely to array-based/vector-based primitives. However, although each column in a database (DB) table contains homogeneous data, different columns may contain different base types. Thus, we have used some insights from array languages like ELI [14], Q [1], and Q'Nial [29] to include some specific advanced types for dictionaries and tables which provide a clear mechanism to represent the DB tables, while still being amenable to optimizations.

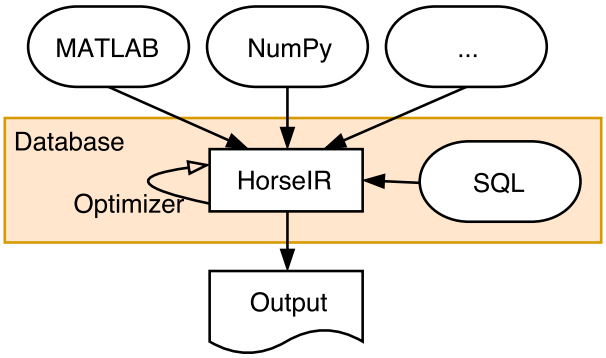

Figure 1: Overview of HorseIR approach, providing a common IR for SQL and procedural language UDF interpreters.

As the core data structure in HorseIR is vector (corresponding to columns in the DB tables), we have provided HorseIR with a rich set of well-defined built-in functions. They have clearly defined semantics, are easy to optimize, and in the case of element-wise built-in functions, are easy to parallelize. We have also carefully considered the type system for HorseIR, allowing for declaring variables with explicit types, as well as providing a wild-card type and type inference rules.

To demonstrate the feasibility of our approach we compared the standard version of a popular open source column-store RDBMS, MonetDB [25], against optimized HorseIR code for 10 queries from the standard TPC-H SQL benchmark set [45]. An important insight in doing this comparison was that it is important to leverage the SQL optimizations before performing the generation and optimization of HorseIR. Thus, we generate HorseIR from the optimized execution plan computed by MonetDB. These execution plans are generated from the SQL queries and are optimized based on the operators in the query, and the size and structure of the input dataset. This approach allows us to benefit from all of the database-specific optimizations, as well compiler-based optimizations, thus leveraging the years of optimization development that has been done in both the database and compiler communities. In fact, our performance results show that the HorseIR approach outperforms the standard MonetDB approach by 101%.

We also demonstrate the benefits of optimizing across the UDF/SQL by using the Black-Scholes benchmark [9] as a UDF and looking at 10 variations of an SQL query which calls the UDF. Since the HorseIR optimizer can optimize interprocedurally, the context of the SQL query can be used to perform optimizations that would not be possible if the UDF and the SQL query were optimized

[1]The term working dataset is used to denote the copy of data that has been brought from the disk to main memory for processing a request.

separately.

The main contributions of this paper are that we:

- identified the need for a common IR for SQL queries and UDFs;
- designed and implemented an array-based IR, called HorseIR, to represent both SQL queries and UDFs;
- demonstrated how to leverage the many optimizations developed by the database community in terms of generating efficient execution trees for declarative queries depending on the data set;
- identified the important compiler optimizations for generating efficient HorseIR code from SQL query plans;
- identified important cross-boundary optimizations between SQL queries and UDFs; and
- demonstrated performance benefits of the approach, as compared to MonetDB, a popular column-store based RDBMS.

The rest of the paper is organized as follows. We first provide the relevant background about RDBMS and array programming in Sec. 2. We then present the details of the design and implementation of HorseIR in Sec. 3; the translation from SQL and source language to HorseIR in Sec. 4; and a set of compiler optimizations in Sec. 5. Finally, we provide our experimental evaluation in 6, related work in Sec. 7 and conclusions in Sec. 8.

2 Background

2.1 Traditional Database Optimizations

The theoretical foundation of SQL is based on relational algebra. RDBMS usually parse a SQL query into a relational algebraic representation [12], as the latter has been known to be easier to optimize [43]. Relational algebra operators are unary or binary, in the sense they accept one or two tables [2] as input. Their output is always a single table. Therefore it is easy to chain these operators into operator trees where the output of one relational operator serves as the input of another operator in order to solve complex SQL queries. Each of the operators can be implemented in various ways. Which one is the most efficient depends on the location in the tree and the input data.

Thus, modern RDBMS optimizers have a query re-write subsystem that generates semantically equivalent relational algebra operator trees [26, 28]. Execution plans are then generated for each of them, which differ in the implementations they choose for individual operators in the tree. The overall cheapest execution is then chosen, whereby the cost model has traditionally focused on I/O cost as execution was assumed to be I/O bound [28].

[2] In relational algebgra, the term *relation* is often used instead of table, but these are synonymous in the context of our discussion.

A key optimization strategy is to try to minimize the number of records that "flow" from one operator to another as part of the query execution plan, as this translates to overall I/O reduction. Thus, *highly selective* operators, that is, operators that output few records compared to the number of input records they receive, will likely be executed first.

However, as main memory has become cheaper and cheaper, RDBMS have now often so much main memory available that the working dataset of a query can completely fit into main memory. With this, RDBMS workloads have started to become compute bound, often not leveraging the underlying processor/memory architecture efficiently [7, 11]. This has resulted in the exploration of "block optimization" techniques [38, 47] to leverage CPU performance, with [10] being more sophisticated in their approach by accounting for CPU cache utilization.

2.2 UDFs & Embedded Interpreters in RDBMS

The notion of a UDF was first introduced in [33]. The general idea was to provide the user with a way to leverage a conventional procedural language to write code snippets for computations that cannot be expressed via SQL. The growing popularity of new generation interpreted languages such as Python and R [20], especially among data scientists[3], encouraged RDBMS vendors to embed them in the RDBMS software [41, 36, 44, 24, 35]. However, UDFs can work only on data that is already in main memory unlike the execution of SQL queries where data can be retrieved from disk should it not already reside in main memory. Therefore the RDBMS engine performs the disk to main memory data transfer before invoking UDFs on the data.

Although convenient, in many aspects this integration of procedural languages into an RDBMS engine is done less than optimally. The RDBMS query engine is concerned with only optimizing the data management operations as required to support a SQL query. It has no insight into the internal working intrinsics of a UDF. Any optimizations within a UDF is left to the language interpreter. Often physical data format conversions are required between the two systems, although the logical aspect of the data structure is the same. Hence, these two systems, while residing in the same component and executing parts of the same request, treat each other like black boxes, eliminating the possibility of optimizing the request as a whole.

There has been some work done in ameliorating this gap, most notably in column-stores. By choosing the physical structures to match that of the interpreter with minimal pre-processing, [41, 36] have attempted to reduce the need for physical data conversions in MonetDB. [24] applies a similar approach on SAP HANA [22], a commercial RDBMS.

2.3 Array Programming Overview

Array programming is supported by a wide range of programming languages, such as APL, MATLAB, and FORTRAN 90. The main characteristics of array programming are as follows. 1) Array objects are the main data structure. An array object is able to represent an arbitrary dimensional array. As a consequence, programming with arrays comes with succinct and expressive code; 2) Array programming languages provide a rich family of operators as built-in functions. The fundamental idea of array programming is to apply an operation on all items of an array without an explicit loop. If an operation is mappable, the code can be executed in parallel on each item of the

[3]PYPL PopularitY of Programming Language Index `http://pypl.github.io/PYPL.html` as of October 2017.

array. For example, MATLAB's element-wise functions are well tuned for implicit data parallelism.

A special case of array programming is vectorization. It can take place in either low-level hardware or high-level programming languages. Modern hardware is actively adopting the concept of vectorization in their chip design. For instance, Intel Advanced Vector Extensions (AVX)[4] is an instruction set designed for efficient vector operations. One instruction performs one operation on multiple data items simultaneously. Another kind of vectorization is the source-level translation from a scalar form to a vectorized form to reduce the overhead of explicit loop iterations [34, 15].

HorseIR is influenced by ELI, Q and APL. It is designed with a set of special built-in functions. Other than a general array object, it introduces homogeneous vector as a basic type. Moreover, it provides a set of list based types, including a table type, for handling heterogeneous data. Compared with conventional array programming, our design simplifies the complexity of language semantics, while keeping the flexibility of handling complex data structures.

3 Horse IR Design and Implementation

The design of HorseIR was motivated by the need to capture a very clean array-based IR, with clear semantics which enables optimizations and automatic parallelizations. The IR also needs to handle both array-based computations from UDFs, and computations from SQL queries, in a unified manner. In this section, we provide a brief introduction to HorseIR by giving the key program structures, types, shapes and implementation details.

At its core, HorseIR is a typed, 3-address, SSA-based intermediate representation with: a simple module system; static scoping; call-by-value semantics; a rich set of base types including a wild-card type; key compound types including arrays, lists, dictionaries, tables and keyed-tables; and a rich set of well-defined primitive array operations.

3.1 Program Structure

Modules

A valid HorseIR program consists of a set of modules, with each module defining zero or more static fields and zero or more static methods. For example, Listing 1 shows an example module, `BigDiscount`, with two methods. If a module contains a method called `main`, then this can be used as an entry point of a program. In the `BigDiscount` module there is a `main` method that reads from the database table `lineitem`, loads the column `l_discount`, and then calls the method `compute_discount`.

In addition to the programmer-defined modules, there is also a pre-defined `default` module which collects any fields or methods that are defined outside of a module. Further, a module can import one or more methods from another module using an `import` statement. Imported methods may be called using the name of the method (without the name of the module), except in the case where there is a conflict between a method being imported and the current module, in which case the method must be called with the module name explicitly.

With this simple module design, HorseIR provides a mechanism for modularizing complex software and provides a flexible way of specifying reusable libraries.

[4] `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`

Listing 1: An example HorseIR module

```
module BigDiscount {
  import Builtin.*;  // import all builtins
  def compute_discount(discount:f64):i64 {
    // find all discounts greater than 50%
    t1:bool = @gt(discount,0.5:f64);
    // count how many discounts greater than 50%
    t2:i64 = @sum(t1);
    return t2;
  }
  def main():i64 {  // an entry method
    // load table: lineitem
    a0:table = @load_table(`lineitem);
    // load column l_discount from table
    t1:f64 = check_cast(
              @column_value( a0,`l_discount),f64);
    // call method compute_discount
    t2:i64 = @compute_discount(t1);
    return t2;
  }
}
```

Methods

A method has zero or more parameters and 0 or 1 return values. Parameters are passed by value, which simplifies program analysis, but also means that copy-elimination is an important optimization, as in the case of efficiently executing MATLAB [23]. Method calls preceded by the `@` indicate user-defined or library method calls, whereas those without the `@` are system calls such as `check_cast`. Methods may be overloaded, but the type signatures of overloaded methods must not allow for any ambiguous invocations.

Static Fields and Local Variables

A static field has the scope of its defining module, and local variables have the scope of their enclosing method. Each static field and local variable must have a declared type, although that type may be the wild-card type.

3.2 IR Types

Deciding on the type system was a very important decision in the design of HorseIR. The first key decision was that HorseIR should be statically typed, but with a special wild-card type that allowed for the case when a static type could not be determined, thus indicating where a dynamic type check must be made. This tension between static and dynamic typing is partly due to the fact that many common array languages are dynamically typed, and so it would, in general, not be possible to generate statically-typed HorseIR from those languages. On the other hand, the database tables have declared types, and so generating statically-typed HorseIR from queries is possible, and is preferred. Finally, it has been well established that static types and shapes can lead to much more efficient array-based code, so one should aim for as much static typing as possible [32].

Base Types and Homogeneous Arrays

A second key decision was to support quite a rich set of base types which includes: (1) all of the base types you would expect to find in an array-based language (boolean, char, short, int, long, float, double, and complex); (2) additional base types that you would expect to from an SQL database (string, month, date, time, datetime, minute, and second); and (3) a special type, symbol, which provides for an efficient representation of immutable strings which is very important for efficient in-memory database representations.

An underlying principle in array-based languages is that many built-in operations are defined over homogeneous arrays. Since each homogeneous array can be stored in a contiguous memory region, it is a cache-friendly design, as well as being easily partitionable for parallelism. Thus, our declarations actually denote arrays, with each array having an explicit base type, and an implicit extent (number of dimensions) and a shape (the size of all dimensions). Only the base type is declared, but the extent and/or shape may sometimes be inferred. For example, the parameter declaration `discount:f64` in Listing 1 declares that `discount` is a homogeneous array with a base type of **f64**. In this case, a shape inference would be able to determine that it is actually a vector, based on the output shape of the built-in method `column_value`.

Advanced Heterogeneous Data Structures

Although homogeneous arrays are excellent for core scientific computations, the data stored in an SQL database is not homogeneous, but has columns with different data types. Thus, HorseIR was defined with key heterogeneous data types to effectively capture SQL-like data in a manner that interacts well with array-based primitives.

A list type is the fundamental type which provides cells for holding different types and lists can be nested. Other advanced types derive from this nested list type. A dictionary is a list of the pairs of keys and values; a table is a special case of a dictionary which requires each value has the same length; a keyed table serves as the glue for two ordinary tables; and an enumeration keeps a pair of key and value in which the value is replaced with the index of the position of its occurrence in the key.

HorseIR supports many important built-in functions for dealing with these data structures, and four of them are used extensively in the code generation strategies described in Sec. 4.

The function *list* takes an arbitrary number of arguments and returns a list with each argument saved into a single cell. The length of the returned list is the number of arguments. The function *raze* unravels an input list (including nested lists), creating a vector containing all of the leaf elements of the list. Note that *raze* expects all leaf elements to be of the same type, since vectors are homogeneous data structures. Further, HorseIR supports the built-in functions *keys* and *values* for fetching keys and values from a keyed table or a dictionary.

Elementwise Operations for Arrays

Like many array-based languages, HorseIR supports a large collection of elementwise operations which take either one (unary) or two (binary) parameters. An *elementwise unary operation* takes one argument and maps its operation on each item of the argument. Thus, the shape of the output is the same as the shape of the input. An *elementwise binary operation* takes two arguments. If neither of the lengths of the two parameters is one, they must agree on the length, denoted N. Therefore, there are three possibilities: *1-to-N*, *N-to-1*, and *N-to-N*. The result will always be of length N, with the binary operation being applied in a pairwise fashion on each element (in the case where one argument is a scalar, it is virtually expanded through replication to a vector of length N).

Each Operations for Lists

Since HorseIR also includes list-based data structures, it provides a variety of map-like operations. HorseIR supports one unary operation, `each_item(f,x)`, which applies a function `f` over all elements of list `x`, and three binary operations, `each(f,x,y)`, `each_left(f,x,y)` and `each_right (f,x,y)`. For the binary operations, the i'th elements of the outputs are computed as `f(x(i),y(i))`, `f(x(i),y)` and `f(x,y(i))` respectively.

Other Functions

HorseIR also supports helper functions and system functions. For example, the function *len* returns a scalar which indicates the length of its input argument, and function *load_table* is used to load a database table into a HorseIR table.

3.3 HorseIR Implementation

We have implemented three core components for HorseIR: (1) a HorseIR front-end, which parses, performs semantic checks, and generates an AST; (2) a library of efficient and parallelized implementations for the rich set of built-ins for HorseIR, and an Interpreter for HorseIR.

HorseIR Front-end

The HorseIR front-end is built with ANTLR4 [39] which is a popular parser generator. The tool supports Unicode encoding in strings which is useful because Unicode encoding widely exists in databases. We defined a clean grammar for HorseIR, and we developed and implemented a transformation from the CST delivered by ANTLR to an AST suitable for optimization and interpretation. The front-end also performs the type-checking and semantic analysis, replacing wild-card types with inferred types whenever possible.

Built-in-function Library

HorseIR employs a single-function-multiple-implementation strategy to embrace the various kinds of data from database systems. One built-in function may have one or more implementations that are specialized to the correct base type, or the size or shape of the input data.

Since elementwise operations have no data dependencies, the HorseIR library provides a parallelized version for all of them. Moreover, HorseIR also supports parallel code for other frequently used operations. For example, the operation `sum` is implemented with the aggregation of partial sums from each parallel thread. Currently, we use OpenMP to implement parallel code in the library and it is also convenient for generating efficient parallel code from these well-defined high-level built-in functions. The design of HorseIR enables simple parallelization and exposes more information to subsequent optimizations because we support simple and clear combinations of vector/arrays and lists.

However, parallelizing a single function is not sufficient because the loop barrier introduced for each operation is expensive. Thus, it is often beneficial to merge two or more functions using loop fusion. HorseIR exposes such fusion using a function called `fuse_op` which takes as arguments a list of primitive functions, and a list of arguments. This instruction corresponds to specialized fused code in the back-end.

HorseIR Interpreter

The HorseIR interpreter uses a standard interpreter design, it directly interprets the input Hor-

seIR, making calls to the built-in library. Interpreting (rather than compiling) in this context is quite effective since the code generated for queries is quite small and the code makes good use of the efficient built-in library functions. In the future, we may add some JIT functionality to the interpreter in order to further gain from specialized code.

4 Compiling to HorseIR

RDBMS optimizers are able to generate very efficient execution plans that take the type of operators in the query and their selectivity into account, as well as the size and structure of the underlying tables. In order for our HorseIR approach to take advantage of these optimizations, such execution trees must be translated into HorseIR. In this section, we first describe how individual relational algebra operators can be expressed in HorseIR, then discuss the translation of full query execution plans and how they relate to the procedure code, to finally discuss possible limitations.

4.1 Mapping Relational Algebra to HorseIR

In this section, we discuss how the most fundamental relational operators are translated into HorseIR.

Projection $\Pi_{a_1,...,a_n}(R)$ takes the records of table R as input and returns the same records but only the columns of R with column names $a_1, ..., a_n$.[5] In HorseIR, the function *column_value* loads a column given a table name. The column names are formed into a string or symbol vector. Then, a new table is returned with the function *table* which takes both column names and values. Let $col_k = @column\_value(T, a_k)$ where $k \in [1, n]$. Thus, we can have the project operation in HorseIR as follows.

```
columnName = (a_1,a_2,...,a_n);
columnVal  = @list(col_1,col_2,...,col_n);
newTable   = @table(columnName,columnVal);
```

Selection is denoted as $\sigma_P(R)$ where P is a collection of selection predicates and R is a table. The selection returns those records of R whose attribute values fulfill the condition P. Formally, $P = (P_1 \ < op > \ ... \ < op > \ P_n)$ where *op* is either $\wedge$ for a logical `AND` operation or $\vee$ for a logical `OR` operation.[6] A predicate returns `True` or `False` when its input data satisfies a specific condition or not.[7] In HorseIR, we need two steps to achieve selection. First, the *op* is replaced with one of two built-in boolean functions *and* or *or*. The function $compress(A, B)$ is defined as $\{B_t \mid A_t = true, \ t \in [1, n]\}$, where A is a boolean vector and both A and B have the same length n. Second, the result of a predicate is a boolean vector, which is applied to a vector to fetch valid records. In the example below, *p1* and *p2* are two boolean vectors and a new vector is returned after filtering data.

```
pred      = @and(p1,p2);
newVector = @compress(pred,vector);
```

[5] Projection refers to the SELECT clause of a SQL query, e.g. SELECT $a_1, ...a_n$ FROM R.

[6] P is represented in the WHERE clause of a SQL query, e.g., WHERE $a_1 < 100$ AND $a_2 = 10$.

[7] While SQL follows three-value logic, our current implementation of HorseIR supports only boolean logic, we will address this in a future work. Our current test scenarios do not require three-value logic.

Join. A join operation takes two tables as input and connects records from the two tables that fulfill certain conditions. Let R_1 be a table with columns $(col_{a1}, ..., col_{an})$, and R_2 be another table with columns $(col_{b1}, ..., col_{bm})$. Then, the join operation returns a new table:

$$R_1 \bowtie_{COND} R_2$$

where $(COND \leftarrow (col_{a1} = col_{b1}) \wedge \ ...)$. The new table contains the columns from both the tables R_1 and R_2. A record r_1 from R_1 together with a record r_2 from R_2 build a record in the new table, if r_1 and r_2 fulfill the conditions $COND$. In HorseIR, we provide a primitive data type, *enumeration*, to keep the relationship between two tables after join. An enumeration takes two parameters K and V. The type and shape of K and V must be the same. Then, the enumeration records the indices of the first occurrence of V's value in K. On the other hand, the target variable K is stored, while the source variable V can be ignored since all the information has been saved into the enumeration. The builtin function *enum* constructs an enumeration. The optimization opportunities in a join are discussed in Sec. 5.

```
newEnum = @enum(K, V)
```

Aggregation. An aggregation function takes a list of values as input and returns a single value as output, such as *sum* (sum of values) and *count*(number of values). A formal definition of aggregation is

$$(G_1, G_2, ..., G_n)\ AGGR\ _{F_1(a_1),F_2(a_2),...,F_m(a_n)}\ (R)$$

where (i) $G_1, G_2, ..., G_n$ is a list of columns (in the table R) to be grouped; (ii) $a_1, a_2, ..., a_n$ are names of columns in R; and (iii) $F_1, F_2, ..., F_m$ are aggregation functions. In HorseIR, the function *group* aggregates values which can be a vector or a list, and returns a dictionary in which a key is the index of the first value in a group and a value consists of the indices of same values in a group. (i.e. $dict < i64, list < i64 >>$). After array indexing with lists, the aggregation functions are applied. As a result, in each cell of a list, it contains a single value after aggregation functions. Finally, it is unraveled with the function *raze* and a vector is returned as the result of the aggregation function on a column (e.g. the function *count* on G_1 in the following example).

```
listG    = @list(G_1,G_2,...,G_n);
dictG    = @group(listG);
indexG   = @values(dictG);
valG1    = @each_right(@index,G_1,indexG);
countG1  = @each_right(@count,valG1);
vectorG1 = @raze(countG1);
```

4.2 SQL Execution Plans

The idea is to take an optimized execution plan generated by the RDBMS (in our case MonetDB) and translate it to HorseIR. The execution plan has a clearly defined order of operators. As just seen in the previous section, each of these operators can be mapped into one or more lines of HorseIR code. Our code generation is similar in concept to a compiler back-end with code patterns. Therefore, translating an execution plan into a base HorseIR program is fairly straightforward.

However, there remains plenty of opportunities to optimize this base program, as there is no one-size-fits-all strategy. For example, the computation of join with an enumeration at run-time is expensive, especially when two tables have a relatively large number of rows. In section 5 we will identify and explore opportunities for further optimizing the generated HorseIR code.

4.3 Translation of High-level UDFs to HorseIR

Since HorseIR is a high-level IR, the language used as a source language for UDFs should be relatively high-level, too, and support array programming, such as MATLAB and Python. In the translation from MATLAB to HorseIR, if the input type information is unknown, it can be set to a wildcard type and its actual type is specified in subsequent type checking. When HorseIR starts with embedded queries, it is able to collect the type information of columns from database systems. By knowing the type information, HorseIR can emit efficient code by removing type guards after type inference.

One possible approach to translate MATLAB code to HorseIR is using McSAF. McSAF is a static analysis framework developed at the Sable lab for MATLAB [21]. In our future work, we will consider converting standard MATLAB programs into HorseIR by using such existing infrastructures to make this process automatic.

4.4 Limitations

Benefits of compiling to HorseIR may be limited to high-level array based languages. Even in an array-based language which supports explicit loops (e.g. for-loops in MATLAB), it requires vectorization techniques to detect possible source-to-source translation [15]. Such automatic vectorization techniques are usually inefficient and are greatly affected by programming styles. HorseIR supports branches that enable it to handle loops. However, it can become a performance bottleneck. For best performance benefits, UDFs should be written in array form.

5 Optimizing Horse IR

In this section, we present our optimization strategies for HorseIR programs.

5.1 Intraprocedural Optimizations

We first look at optimization opportunities within a method. Classic compiler techniques, such as type and shape analysis and data dependency analysis, are necessary for understanding how a program behaves. Database system researchers also realized the importance of these techniques and integrated some of them into their database systems, such as MAL optimizer in MonetDB [25]. This is a valuable lesson we need to take in building our own optimizer. We have found the following optimizations could be used in the context of optimizing HorseIR, and in particular HorseIR that is generated from SQL queries:

Constant propagation and folding (CPF). This is a standard technique to propagate and pre-compute constant values, especially for date and time in HorseIR.

Common sub-expression elimination (CSE). Redundant common sub-expressions are replaced with a common variable which is computed only once before any common sub-expression.

Strength reduction (SR). An operation is rewritten into a more efficient operation. For instance, when a boolean vector multiples a real vector, the expensive multiplication operation can be replaced with a cheaper ternary operation as a conditional expression.

Loop fusion (LF). Since each array-based builtin function is parallelized separately, each function has a synchronization barrier to collect result from all spawned threads. Adjacent function calls can be fused, reducing the number of synchronization barriers. We found that this sort of loop fusion is particularly useful for adjacent logic operations performed on boolean vectors. This pattern is common in HorseIR code generated from the *WHERE* clause in SQL.

Peephole optimizations (PH). A peephole is a code pattern which compiler can recognize within a code snippet, then rewrite it into more efficient code. We report a couple of peepholes as follows:

- *Groupby and orderby.* A *groupby* is an aggregation operation. Its input may take either a vector or a list since it may operate on multiple columns. Its implementation first sorts the input and later look for the same items from neighbours. If both *groupby* and *orderby* operate on the same columns, the additional sorting in *orderby* should be removed.
- *Inner self-join.*[8] When only aggregation operations occur right after an inner self-join, the join can be replaced with a *groupby* which is cheaper than join.

Join transformation (JT). This technique intends to reduce the overhead of reconstructing a new join after filters. In RDBMS, a join can be pre-computed by using *keys* and *foreign keys*.[9] The result of a join is stored in an enumeration, `ev`, with a pair of <key, fkey>. Except for the joins which must have to be generated on-the-fly, the pre-built join is able to be optimized in the following scenarios:

1. When a key is selected with a filter, its foreign key gets a boolean mask by indexing through indices stored in the enumeration `ev` before being compressed;
2. When a foreign key is selected with a filter, its key stays the same. We only need to update `ev` to `ev′` with less items in its fkey part.
3. When a key and its foreign key both are selected, the relation can be updated in the following steps: (i) update fkey to fkey′in scenario 2; (ii) update key to key′and fkey′to fkey″in scenario 1; and (iii) return a new enumeration with <key′, fkey″>.

5.2 Interprocedural Optimizations

Since the RDBMS optimizer and UDF interpreter treats each other like black boxes, it often becomes the programmers' responsibility to optimize the interaction between the two and is often not feasible. We adopt the concept of interprocedural optimizations to optimize HorseIR programs holistically. When two programs from different language systems (e.g. SQL and NumPy) are translated to a single HorseIR program, we can employ interprocedural optimizations over the combined IR representation.

[8] A self-join is special case of the join operation when a table is joined with itself, often on different columns. An example would be joining an `employee` table with itself over the columns `employeeid` and `managerid` to find who is the manager of an employee.

[9] A column can be a key in a table, if its value is unique for each record in the table (such as `deptid` in `department` table. When another table, such as `employee` tries to relate with the `department` table, `deptid` becomes a foreign-key column in the `employee` table.

UDF vectorization (UV). A UDF is called from a query is similar to a function called from a loop. Prior work in MATLAB showed the vectorization on loops with functions was promising to improve the overall performance [15].

Interprocedural program slicing (IPS). Interprocedural program slicing is able to trace for a variable `x` to collect all statements that may affect the value of `x` between two points. For example, a query may pass many input columns to a UDF, of which some may not be used. IPS can eliminate the unnecessary computation in such non-traced statements.

The UDF benchmark in Sec. 6.3 is designed to demonstrate how the aforementioned optimizations impact queries with UDF. The list of interprocedural optimizations should grow as we further explore and learn more about queries with UDFs.

6 Evaluation

In this section we present the results of our test evaluation comparing HorseIR with MonetDB conducted using the TPC-H SQL benchmark and the Black-Scholes UDF benchmark.

TPC-H [45] is a widely used standard benchmark suite intended for testing the performance capabilities of RDBMS implementations meant for large scale data processing. This suite has also been used in the past to benchmark implementations on MonetDB [10]. TPC-H mimics a Business to Consumer (B2C) database application[10] and has 8 tables. It also contains a suite of SQL queries, from simple to quite complex covering a wide spectrum of SQL constructs. Being a SQL benchmark, these queries are void of any UDF usage. The suite also contains a data generator that can create synthetic data sets over a variety of sizes, known as the *scale factor* (SF) of the database. A scale factor of 1 corresponds to a database size of approximately 1 GB, with higher scale factors proportionately increasing the size of the database.

For testing UDF performance optimizations, we choose to implement the Black-Scholes algorithm from the PARSEC benchmark suite v3.0 [9]. Black-Scholes is used in finance to compute the price variation of European options over time by using a partial differential equation (PDE). This algorithm is fully vectorizable, and can be efficiently written using array programming. As the original implementation[11] is in C, we implemented a Python module using NumPy UDFs for integration with MonetDB. We use the random data generator from the PARSEC package to generate one million records. In order to use MonetDB UDFs, these records are all stored in one database table `blackScholesData` consisting of 9 columns.

6.1 General Experimental Setup

The experiments are conducted in a multi-socket multi-core server, `Sable-Intel`, equipped with 4 Intel Xeon E7-4850 2.00GHz (total 40 cores with 80 threads) and 128 GB RAM running Ubuntu 16.04.2 LTS. We use GCC v7.2.0 to compile HorseIR source code with the maximum optimization option -O3; MonetDB version v11.27.9 and NumPy v1.11.2 along with Python v2.7.12 interpreter

[10] Documented in page 13 of the TPC-H benchmark description available at http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf

[11] The complete PARSEC package is downloadable here http://parsec.cs.princeton.edu/

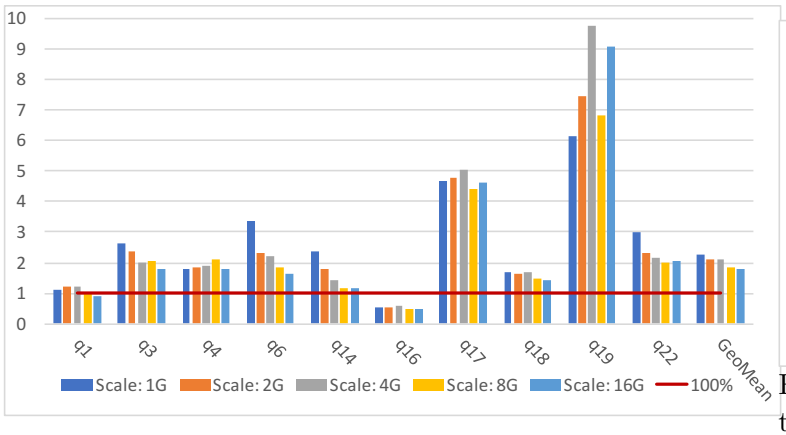

Figure 2: The ratio of MonetDB to HorseIR in execution time (ms)

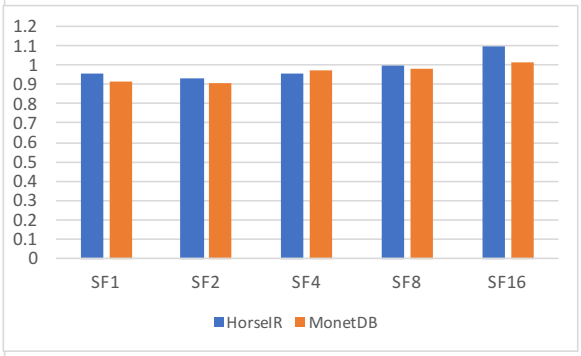

Figure 3: The geomean of the ratio of the response time of HorseIR and MonetDB with thread 20 / 40 over different scale factors

for UDF support in MonetDB. Both systems are set to run using 40 threads by default. The response time is measured only for the core computation, and does not include the overhead for parsing SQL, plan generation and serialization for sending the results to the client. We only consider execution time once data resides in main memory. For MonetDB we guarantee this by running each test 15 times but only measure the average execution time over the last 10 times. After the first 5 runs response times stabilizes showing that all data has been brought from disk to main memory by then. Scripts and data used in our experiments can be found in our GitHub repository[12].

6.2 TPC-H SQL Benchmark

In this section we present the results for 10 of the 22 SQL queries of the benchmark (1, 3, 4, 6, 14, 16, 17, 18, 19, and 22). These queries have been selected to cover a variety of performance impacting dimensions such as number of joins, condition predicates, sizes of the tables, number of columns and records that are returned, etc. For each of the queries, we took the execution plan generated by MonetDB's optimizer and translated it manually, following the translation and optimization techniques detailed in sections 4 and 5 to HorseIR. Further, we verify the scalability of the implementation by testing on databases of varying scale factors SF 1, 2, 4, 8, and 16.

Fig. 2 shows the ratio of the execution time with MonetDB to the execution time with HorseIR. For most queries, HorseIR outperforms MonetDB. As can be seen from the geometric mean over all queries, MonetDB takes roughly double the time to execute than HorseIR, across all the database sizes tested. The improvements achieved by HorseIR range from 124% at SF 1 to 79% at SF 16.

The compiler optimizations that are used for each of the queries, and which are the cause for these improvements are summarized in Table 1. We can see that for different queries, different optimizations provide benefits, and for some queries these optimizations provide very significant performance improvements. This emphasizes the need to have a diverse arsenal of techniques, as the optimization opportunities, and their potential impact can vary significantly based on the characteristics of each query. For example, our observation is that q17 benefits significantly from peephole optimization, getting up to 361% improvement for SF 16 whereas q19 shows improvement of up to 804% from loop fusion optimization. Although not to the same extent, q3 and q4 benefit

[12]link removed for double blind review

Table 1: HorseIR optimizations for TPC-H queries.

Query	Lines of Code		Optimizations					
ID	SQL	HorseIR	CPF	CSE	PH	SR	LF	JT
q1	21	80		✓	✓		✓	
q3	22	77			✓			✓
q4	21	57	✓				✓	✓
q6	9	40	✓				✓	
q14	13	46	✓	✓		✓	✓	
q16	30	79			✓			✓
q17	17	46			✓		✓	
q18	33	77					✓	✓
q19	35	111	✓				✓	
q22	37	63					✓	

from join transformation optimizations by around 103%.

Table 2: Difference of response time (ms) between different scale factors (SF) in HorseIR (H) and MonetDB (M).

	$\Delta_{SF2-SF1}$		$\Delta_{SF4-SF2}$		$\Delta_{SF8-SF4}$		$\Delta_{SF16-SF8}$	
ID	H	M	H	M	H	M	H	M
q1	164.9	213.0	215.3	270.5	553.4	430.3	1204.6	925.3
q3	56.3	121.1	120.3	200.5	226.5	479.2	536.9	839.9
q4	16.3	31.2	19.7	39.1	49.0	117.2	146.3	225.8
q6	18.2	27.9	19.7	38.3	70.9	114.6	122.2	170.2
q14	22.1	28.1	42.1	44.6	108.6	101.5	177.8	209.4
q16	146.7	82.2	303.4	200.2	1017.9	431.8	1642.3	858.1
q17	29.1	142.9	65.8	345.2	158.7	615.3	289.2	1397.2
q18	164.8	264.5	320.7	572.0	698.1	879.6	1412.0	1959.2
q19	24.4	208.5	34.5	436.1	120.6	588.1	126.2	1590.1
q22	33.3	62.4	61.4	120.5	143.0	271.3	291.3	612.4

By analyzing the benefits of HorseIR optimization techniques across the queries tested, we observe that loop fusion, which helps queries that contain filter conditions, is the most commonly used optimization. Constant propagation and folding, strength reduction, and common sub-expression elimination opportunities were moderate, given the nature of the queries. While four queries benefit from peephole optimization, this is still a category that we are exploring and hope to further extend in the future.

To get a better understanding of the impact of database size, Table 2 presents the delta increase in response time for each query as we change from one scale factor to the next higher one. While the cost of all queries increase for both HorseIR and MonetDB as the database size increases, we can clearly see that for the majority of the queries MonetDB's response time increases a lot more than HorseIR's, highlighting the cost benefits of HorseIR's optimization techniques as data sets scale.

Another observation is that MonetDB's performance with respect to HorseIR is slightly worse off at lower scale factors compared to larger scale factors. To investigate this we ran the experiment

by reducing the number of threads used to 20. Figure 3 shows the geometric mean of the ratio of execution with 20 threads compared to 40 threads over all queries. We can see that both MonetDB and HorseIR are better off with 20 threads at SF 1, but as the database size increases, at SF 16, having 40 threads is faster for both the systems. This is not unexpected, as for smaller data sets the overhead due to parallelism can outweigh its benefits. However, what is interesting is that for smaller data sets, HorseIR seems to exhibit much less overhead from having extra threads compared to MonetDB. On the other hand we can see that for SF 16, the speedup of HorseIR is higher than MonetDB when increasing the number of threads from 20 to 40. Therefore we can conclude that HorseIR can scale elegantly with a varying number of threads and data set sizes.

Our conclusion from the analysis of the TPC-H benchmark results is that the unified approach of HorseIR is very promising. As it uses as its basis the same optimized execution trees as MonetDB it is able to leverage the same database optimizations in terms operator sequence and data flow. On top of that, it exploits additional compiler optimizations which were tailored to the array-based approach of HorseIR, thus providing an overall faster execution.

However, HorseIR has some performance gaps in certain operations compared to MonetDB. For example, our analysis of q16 where HorseIR performs worse than MonetDB indicated that this is due to the low performance in the current integration of HorseIR with the PCRE[3] pattern matching library that we use in evaluating predicates which require searching for patterns in text data. We are working on improving this integration by exploring just-in-time (JIT) compilation techniques.

6.3 Benchmarking UDF Optimizations

In addition to experimenting with pure SQL queries, as in the previous section, we also wanted to see how the integrated HorseIR approach compared the UDF approach currently available with MonetDB.

For the HorseIR version, we merely expressed the array-based algorithm for Black-Scholes as HorseIR, and combined it with the HorseIR code for the query, thus having both the UDF and query in the same IR.

For the MonetDB version, we expressed Black-Scholes as a Python module, and we wrote the appropriate SQL query to connect to the Python code. The SQL query first selects all the columns from this table Block-Scholes input data table and passes it to the wrapper UDF defined in query. The wrapper UDF then invokes the Black-Scholes algorithm in the Python module, which computes the result (*option price*) and returns it to the wrapper UDF.

In order to test the two different types of UDF programming approaches that databases support, we created two variants of the UDF. In one variant, we created a *scalar UDF* that returns just the computed `optionPrice` to the calling SQL. A scalar UDF consumes one or more columns as its input and produces exactly one column as output. For each input row, it produces one output row. Therefore, scalar UDFs are desirable when implementing computations that are embarrassingly parallel.

```
CREATE SCALAR UDF bScholesUDF
(spotPrice, ..., optionType)
{
 import blackScholesAlgorithm as bsa
 ret bsa.calcOptionPrice
          (spotPrice, ..., optionType)
};
```

Next we implemented the solution as a *Table UDF*, which returns in table form the computed `optionPrice` along with the associated `spotPrice` and `optionType` which are columns from the original input table.

```
CREATE TABLE UDF bScholesTblUDF
(spotPrice,...,optionType)
{
 import blackScholesAlgorithm as bsa
 optionPrice = bsa.calcOptionPrice
          (spotPrice,...,optionType)
 ret [spotPrice,optionType,optionPrice]
};
```

In order to have a broad set of tests and comparisons, we first integrated these two UDF versions into a straightforward base query. From there we created three significant variations of this base query. Further, for each of the variation we modified the values associated with the conditional predicates in the selection (WHERE clause), so that selectivity varies between high, low and medium. In a highly selective condition only few of the input records fulfill the condition and thus are in the output result. A query with low selectivity returns most of the input records. Thus, our entire test case consists of 10 queries.

```
-- Base query, bs0_base, Scalar UDF
SELECT spotPrice, optionType,
  bScholesUDF(spotPrice,...,optionType)
  AS optionPrice
FROM blackScholesData;

-- Base query, bs0_base, Table UDF
SELECT spotPrice,optionType,optionPrice
FROM bScholesTblUDF
  ((SELECT * FROM blackScholesData));
```

The base query `bs0_base` selects all the data from the database table and passes it to the UDF and returns all the data produced by the UDF.

The first variation `bs1_*` applies a predicate condition on `spotPrice`, a column which is actually part of the input database table. The objective of this test case is to analyze if the systems can intelligently avoid performing the UDF computation on records that will not be in the result set, that is by first discarding records from the input that do not fulfill the predicate condition and only execute the UDF on the records that qualify. In contrast, a system following an inefficient approach will first compute the UDF over all the input records before applying the predicate.

```
-- Query, bs0_high, Scalar UDF
SELECT spotPrice, optionType,
  bScholesUDF(spotPrice,...,optionType)
  AS optionPrice
FROM blackScholesData
WHERE spotPrice<50 OR spotPrice>100;

-- Query, bs0_high, Table UDF
SELECT spotPrice,optionType,optionPrice
FROM bScholesTblUDF
  ((SELECT * FROM blackScholesData))
WHERE spotPrice<50 OR spotPrice>100;
```

In the next variation, `bs2_*`, the SQL does not include the computed column `optionPrice` in the final result. A smart system should be able to analyze the semantics of the request and avoid processing the UDF all together.

The last variation, `bs3_*` applies a predicate condition on `optionPrice`. As this is a column computed by the UDFs, systems have to process the UDFs across all input records before discarding records that do not qualify, providing little room for optimization.

A major problem that we encountered for our comparison is that MonetDB does not parallelize the execution of table UDFs. Furthermore, although MonetDB is, in principle, capable of executing scalar UDFs in parallel through vectorization, we observed during our experiments that a limitation in the RDBMS prevented it from parallelizing the UDF execution whenever the UDF required more than 3 columns in its input. Hence, in the interest of fairness we decided to also run MonetDB and HorseIR using a single thread for UDF testing. For the sake of completeness and to demonstrate HorseIR's scalability, we have also included metrics for parallel execution of HorseIR using 40 threads.

Table 3: Result of queries with variations on table and scalar UDFs in Black-Scholes, including selection ratios (%) and execution time (ms).

Query ID	Selection	MonetDB (ms)		HorseIR (ms)	
		1 th.	1 th.	1 th.	40 th.s
		Table	Scalar	IR	IR
bs0_base	100.0%	907.1	716.3	664.0	108.9
bs1_high	0.2%	912.1	6.7	16.3	6.4
bs1_med.	50.9%	914.6	349.3	351.5	77.6
bs1_low	99.8%	917.2	699.1	695.7	149.7
bs2_high	0.2%	916.4	4.1	6.5	0.9
bs2_med.	50.9%	915.1	13.4	10.3	5.2
bs2_low	99.8%	915.9	15.3	13.9	6.6
bs3_high	1.2%	914.8	738.6	676.0	109.5
bs3_med.	49.5%	915.0	733.7	688.7	117.7
bs3_low	98.8%	917.4	747.5	683.1	118.5

Table 3 shows the response times for the various queries for the table and scalar UDFs in MonetDB as well as the HorseIR execution times for a single thread and 40 threads. It must be noted that an optimized HorseIR code does not differentiate whether the implementation follows a scalar UDF or table UDF as it is able to translate SQL + UDF requests with the same semantics into identical optimized HorseIR code. We can see that HorseIR performs better than MonetDB for the base

query, attributed to the fact that an integrated IR system does not have the overhead of interfacing between the UDF procedural language interpreter and RDBMS. Both the scalar UDF in MonetDB and HorseIR have comparable performance for the queries `bs1_*` and `bs2_*`. In the case of `bs1_*`, for the SQL using scalar UDF, MonetDB can infer that the conditions are placed on the input column and intelligently discards the records from the input that do not satisfy the condition before processing the UDF, following the traditional database optimization technique of applying high selectivity operations first. HorseIR, using the idea of UDF vectorization, first performs data dependency and side-effect analysis between the input columns and output columns, and is able to apply the predicate on the input column before processing the UDF computation. However, MonetDB is not able to do this optimization for the table UDF, as having no insight into the UDF, it cannot deduce that the output column of the UDF on which the predicate is applied is the same as the input column. This leads to bad performance with table UDF in MonetDB.

For `bs2_*`, we can see that MonetDB is able to do the optimization when the SQL query is using the scalar UDF, avoiding the computation of the `optionPrice` column that is not included in the final result. Similarly, HorseIR, being an integrated system, can apply interprocedural program slicing optimization to avoid computation of `optionPrice`. However, with a table UDF, MonetDB is not able to avoid this computation as there is no way for it to pass this optimization information to the UDF interpreter.

Further, we can see that for `bs3_*`, the response time of the queries are higher than the base query for all the implementations. This is because of the fact that the conditional predicates are applied on the column computed by the UDF, not providing any holistic optimization opportunities. Therefore all the systems are forced to compute the UDF over the entire data set before performing the additional selection.

Finally, on analyzing the impact of selectivity of the predicates, we can see that both the scalar UDF and MonetDB implementations are able to leverage performance benefits from highly selective queries (except for `bs3_*`, where the cost of computing UDF for all the input records is the dominant factor). On the other hand, the lack of a holistic optimization results in the table UDF approach in MonetDB having high costs throughout all test scenarios.

To conclude our analysis on evaluating UDF optimizations, we observe that while there might be an optimal way of implementing some solutions in the current RDBMS + UDF architecture, there are situations where an RDBMS will not have any insight into the UDF to optimize the overall execution. HorseIR, by employing its intra- and inter-procedural optimization techniques can address this gap by being a common IR translation for both SQL execution plans and UDF procedural logic.

7 Related Work

The importance of leveraging compiler optimization techniques in improving the performance of database query processing has gained traction in recent years [30]. A popular approach has been to use query compilers that can compile SQL to low-level programming languages, as offloading the portion of computation-intensive code to efficient compiled languages can improve performance over a native interpreter.

A formal method for constructing a query compiler in Scala is introduced in [42]. Among the works that use IRs for translating from SQL to C code. With a single array-based IR, HorseIR is able to

handle SQL semantics with less overhead compared with the multi-IR design.

Hyper shows a more practical method to improve the query performance by compiling SQL to LLVM to utilize LLVM's compiler optimization infrastructures [37]. TUPLEWARE [19, 18] focuses on optimizing UDF centric workflows. They compile the UDF into LLVM and use distributed programming across clusters for performance. DBToaster targeted high-performance delta processing in data streams by compiling SQL to C++ code [6, 31].

The three systems above relied on a compiler for optimizing generated code, LLVM or C++. The compiler which is good at optimizing procedural code, but knows little about what a query does from a high-level. HorseIR is able to optimize queries with a relatively high-level view. The use of UDF is no longer a black box since HorseIR can see what the functionality of UDF is with proper static analysis.

Zhang et al. introduced array-based extensions in SQL so that enabled database systems to support complex applications, such as the Conway's Game of Life application [46]. Ching and Da [16] implemented an idea of loop fusion for generating efficient parallel code from APL's array-based primitives. HorseIR is able to represent SQL queries in an array form, while adopting optimization techniques from array programming.

KDB+/Q [1], which was adopted in the financial domain, provided a notable approach by fusing SQL and programming languages. The database system KDB+ was implemented in a general-purpose array programming language Q which is an interpreter-based language. Moreover, it supplied SQL interfaces which was a kind of wrapper on top of the language Q. The system internally maintains database systems, while seamlessly supporting an array programming language. However, its interpreter-based design heavily relied on hand optimizations other than systemic compiler optimizations.

8 Conclusion & Future Work

In this paper we review the optimization challenges facing traditional RDBMSes which are now faced with very large memory computers, and increasingly complex queries with complex analytics via embedded procedural language interpreters. We postulate that these challenges are something that the compiler community is well placed to address with its decades of experience in programming language optimization techniques. We proposed HorseIR, an integrated array-based intermediate representation that can be used to represent both the SQL execution plans as well as the UDF code. Our HorseIR implementation takes the SQL execution plan from MonetDB RDBMS and applies a variety of compiler optimization and specialization techniques which were tailored to: (1) the context of the array-based HorseIR, (2) HorseIR's primitive functions, and (3) the kind of HorseIR code generated from SQL plans. By applying inter- procedural optimization techniques, our HorseIR approach is also able to optimize across SQL and procedural language UDF boundaries, thus filling a void in the current RDBMS optimization capabilities.

Performance results from our empirical studies using TPC-H Benchmark for SQL queries and Black-Scholes for UDF testifies that applying compiler optimization techniques over the RDBMS execution plan provides substantial performance benefits. Further, the results also demonstrate that the multi- threading capabilities of HorseIR is on par with MonetDB when it comes to scalability to process large data sets.

For our future work, we are planning to examine how to automatically generate HorseIR from MATLAB. We also plan to further examine additional techniques for enabling HorseIR primitives to make the right algorithmic choice based on the characteristics of the inputs, and the context in which the primitive is being used. On another front, we hope to explore the possibility of performing JIT compilation of HorseIR with support for both CPU and GPU.

References

[1] KDB+. `https://kx.com/`. [Online; accessed August-2017].

[2] MATLAB Machine Learning Toolbox. `https://www.mathworks.com/solutions/machine-learning.html`. [Online; accessed 01-August-2017].

[3] PCRE - Perl Compatible Regular Expressions. `https://pcre.org/`. [Online; accessed August-2017].

[4] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *Proc. of the VLDB*, 2(2):1664–1665, 2009.

[5] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, pages 967–980. ACM, 2008.

[6] Y. Ahmad and C. Koch. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *PVLDB*, 2:1566–1569, 2009.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs On A Modern Processor: Where Does Time Go? In *VLDB*, number DIAS-CONF-1999-001, pages 266–277, 1999.

[8] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[10] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005*, pages 225–237, 2005.

[11] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed Characterization of a Quad Pentium Pro Server Running TPC-D. In *International Conference on Computer Design*, pages 108–115. IEEE, 1999.

[12] S. Ceri and G. Gottlob. Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *Transactions on Software Engineering*, (4):324–345, 1985.

[13] D. D. Chamberlin and R. F. Boyce. SEQUEL: A Structured English Query Language. In *Proc. ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 249–264. ACM, 1974.

[14] H. Chen and W.-M. Ching. ELI: a simple system for array programming. *Vector, the Journal of the British APL Association*, 26(1):94–103, 2013.

[15] H. Chen, A. Krolik, E. Lavoie, and L. J. Hendren. Automatic Vectorization for MATLAB. In *LCPC 2016, Rochester, NY, USA, September 28-30, 2016*, pages 171–187, 2016.

[16] W. Ching and D. Zheng. Automatic Parallelization of Array-oriented Programs for a Multi-core Machine. *International Journal of Parallel Programming*, 40(5):514–531, 2012.

[17] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *ACM*, 13(6):377–387, 1970.

[18] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. An Architecture for Compiling UDF-centric Workflows. *PVLDB*, 8(12):1466–1477, 2015.

[19] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[20] N. Diakopoulos, S. Cass, and J. Romero. Data-Driven Rankings: the Design and Development of the IEEE Top Programming Languages News App. In *Symp. on Comp.+ Journalism*, 2014.

[21] J. Doherty and L. J. Hendren. McSAF: A Static Analysis Framework for MATLAB. In *ECOOP 2012, Beijing, China, June 11-16, 2012. Proceedings*, pages 132–155, 2012.

[22] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD*, 40(4):45–51, 2012.

[23] V. Foley-Bourgon and L. J. Hendren. Efficiently implementing the copy semantics of MATLAB's arrays in JavaScript. In *DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pages 72–83, 2016.

[24] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. *Proc. of the VLDB*, 4(12):1307–1317, 2011.

[25] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two decades of Research in Column-oriented Database Architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 35(1):40–45, 2012.

[26] Y. E. Ioannidis. Query Optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[27] Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). Standard, International Organization for Standardization, Dec. 2016.

[28] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[29] M. A. Jenkins. Q'Nial; A Portable Interpreter for the Nested Interactive Array Language, Nial. *Softw., Pract. Exper.*, 19(2):111–126, 1989.

[30] C. Koch. Abstraction Without Regret in Database Systems Building: a Manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014.

[31] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[32] V. Kumar and L. J. Hendren. MIX10: compiling MATLAB to X10 for high performance. In *OOPSLA 2014, Portland, OR, USA, October 20-24, 2014*, pages 617–636, 2014.

[33] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *VLDB*, pages 294–305, 1988.

[34] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In *ACM SIGPLAN Notices*, volume 35, pages 53–65. ACM, 1999.

[35] T. Miller. Using R and Python in the Teradata Database. White paper, Teradata, 2016.

[36] H. Mühleisen and T. Lumley. Best of Both Worlds: Relational Databases and Statistics. In *International Conference on Scientific and Statistical Database Management*, page 32. ACM, 2013.

[37] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[38] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *International Conference on Data Engineering*, pages 567–574. IEEE, 2001.

[39] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[40] R Core Team. R: A Language and Environment for Statistical Computing, 2014.

[41] M. Raasveldt and H. Mühleisen. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20, 2016*, pages 16:1–16:12, 2016.

[42] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to Architect a Query Compiler. In *SIGMOD 2016, San Francisco, CA, USA*, pages 1907–1922, 2016.

[43] J. M. Smith and P. Y.-T. Chang. Optimizing the Performance of a Relational Algebra Database Interface. *ACM*, 18(10):568–579, 1975.

[44] The PostgreSQL Global Development Group. Procedural Languages. In *PostgreSQL 10.0 Documentation*, 2017.

[45] Transaction Processing Performance Council. TPC Benchmark H, 2017.

[46] Y. Zhang, M. L. Kersten, and S. Manegold. SciQL: array data processing inside an RDBMS. In *SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1049–1052, 2013.

[47] J. Zhou and K. A. Ross. Buffering Databse Operations for Enhanced Instruction Cache Performance. In *SIGMOD*, pages 191–202. ACM, 2004.