HorseIR: Bringing Array Programming Languages Together with Database Query Processing

Hanfeng Chen
McGill University
Canada
hanfeng.chen@mail.mcgill.ca

Joseph Vinish D'silva
McGill University
Canada
joseph.dsilva@mail.mcgill.ca

Hongji Chen
McGill University
Canada
hongji.chen@mail.mcgill.ca

Bettina Kemme
McGill University
Canada
kemme@cs.mcgill.ca

Laurie Hendren
McGill University
Canada
hendren@cs.mcgill.ca

Abstract

Relational database management systems (RDBMS) are operationally similar to a dynamic language processor. They take SQL queries as input, dynamically generate an optimized execution plan, and then execute it. In recent decades, the emergence of in-memory databases with columnar storage, which use array-like storage structures, has shifted the focus on optimizations from the traditional I/O bottleneck to CPU and memory. However, database research so far has primarily focused on CPU cache optimizations. The similarity in the computational characteristics of such database workloads and array programming language optimizations are largely unexplored. We believe that these database implementations can benefit from merging database optimizations with dynamic array-based programming language approaches. Therefore, in this paper, we propose a novel approach to optimize database query execution using a new array-based intermediate representation, HorseIR, that resides between database queries and compiled code. Furthermore, we provide a translator to generate HorseIR from database execution plans and a compiler that optimizes HorseIR and generates efficient code. We compare HorseIR with the MonetDB RDBMS, by testing standard SQL queries, and show how our approach and compiler optimizations improve the runtime of complex queries.

CCS Concepts • **Software and its engineering** → **Compilers**; • **Information systems** → **Database query processing**;

Keywords IR, Compiler optimizations, Array programming, SQL database queries

ACM Reference Format:
Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. 2018. HorseIR: Bringing Array Programming Languages Together with Database Query Processing. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '18), November 6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276945.3276951

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 [16]. Being a declarative language based on relational algebra [11], SQL gives RDBMS implementers the opportunity to optimize the execution plan for a query [7]. In this aspect, one can think of an RDBMS as a dynamic language processor which receives SQL queries as input and dynamically translates the SQL query to an optimized plan that minimizes the execution cost of the query and executes the plan.

Traditionally, these optimizations were targeted at the primary resource bottleneck, I/O [17]. A major RDBMS storage optimization, termed *column-store* architecture [1], where a column's data structure in the disk is akin to that of a programming language array, introduced significant reduction in I/O costs for large-scale data processing. At the same time, the increase in the size of the main memory had improved disk caching, making many database (DB) workloads CPU and memory bound [3, 5].

This had the database researchers looking for new optimization strategies. The "array like" nature of a table's columns in column-store based RDBMSes, especially its working data sets[1], naturally led to a series of studies and optimization strategies focused on the benefits of CPU caching [4, 19]. However, to the best of our knowledge, beyond these

[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.

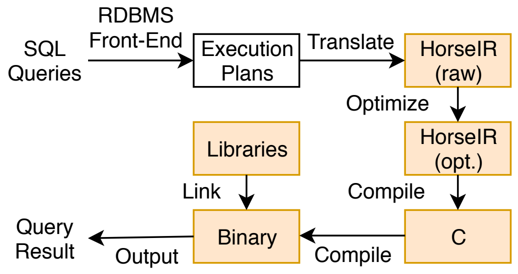


Figure 1. Overview of our approach. Shaded boxes correspond to our contributions.

excursions into exploiting CPU cache-based optimizations, the database community at large is yet to benefit from more comprehensive optimization techniques that the compiler community has amassed from its decades of research on array programming languages. We believe that working datasets 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.

In this paper, we follow a layered approach that facilitates a wide range of compiler optimizations in a systematic way that exploits and further builds upon the many optimizations developed by the database community in terms of generating efficient execution trees for declarative queries. Specifically, we propose an approach where SQL queries are first translated into execution plans using standard DB optimization techniques that consider the operators in the query and the characteristics of the input dataset. These DB optimized plans are the basis on which we explore compiler optimizations. To do so in a systematic way, we translate the conventional database query execution plan into a new array-based intermediate representation (IR), called *HorseIR*, on which compiler optimization strategies are then applied. An array-based IR is a natural choice, given the columnar structure of the working data sets. The modular approach of using an IR spares the RDBMS implementers from having to tailor specific DB operators and algorithms to benefit from various hardware related optimization techniques. Instead, such optimizations can be performed on HorseIR.

The core data structure in HorseIR is a vector (corresponding to columns in the DB tables) and the implementation has a rich set of well-defined built-in functions with clearly defined semantics. They are easy to optimize, and in the case of elementwise built-in functions, are easy to vectorize and parallelize. The type system of HorseIR facilitates declaring variables with explicit types, as well as a wild-card type and associated type inference rules.

Fig. 1 shows the overview of our approach. Queries are first translated to query execution plans. Then, a *plan-to-HorseIR* translator translates the execution plans into raw HorseIR programs. Our current implementation uses the plans produced by the columnar RDBMS, HyPer [19], but the principles of translating execution plans of any RDBMS to HorseIR is largely the same. The HorseIR optimizer first performs standard compiler optimization techniques, such as *dead code elimination* and *loop fusion*. Next, an intelligent pattern-based code generation strategy is deployed to recognize the patterns of HorseIR that have efficient C implementations and replace them with such libraries. Finally, the generated C code is compiled into executable form.

The idea of generating array-based code from SQL is both novel and challenging. To the best of our knowledge, we are the first to introduce an array-based IR that allows column-based in-memory database systems to benefit from a whole range of compiler optimizations. Prior researchers mainly focused on how to generate low-level code directly from SQL queries. Rather than implementing complex operations such as joins directly, HorseIR provides a repertoire of smaller built-in primitives that can be combined to achieve the same functionality. Introducing a high-level IR might open a new research direction for database query optimizations. However, having an additional IR for performing compiler optimizations may result in some overhead and must be considered along with the performance benefits that they bring.

To demonstrate the importance of compiler optimizations and the feasibility of our approach, we performed extensive experiments, using a subset of the TPC-H SQL benchmark [32]. We compare the performance of HorseIR with that of a popular open source column-store RDBMS, MonetDB [14][2]. Execution times are overall relatively similar, showing that an intermediate representation and the use of an array-based programming language is a promising approach that can compete with a highly optimized DB engine. In fact, the optimized HorseIR even outperforms MonetDB. We also analyze in detail as to which compiler optimizations are particularly important for database queries.

The main contributions of this paper are that we:

- identified the need for a common IR for SQL queries;
- designed and implemented an array-based IR, called HorseIR, to represent SQL queries;
- delivered a translator which is able to generate HorseIR code from execution plans automatically;
- identified and applied important compiler optimizations for generating efficient code from HorseIR;
- demonstrated performance benefits of these compiler optimizations, and compared the overall performance with MonetDB, a popular column-store based RDBMS.

The rest of the paper is organized as follows. We first provide the relevant background about array programming and RDBMS in Sec. 2. We then present the details of the design and implementation of HorseIR in Sec. 3; the translation

[2]While we would have liked to compare with HyPer, whose plan generator we used, this was not possible as only the plan generator but not the HyPer execution engine are public. See http://hyper-db.de/interface.html

from SQL to HorseIR in Sec. 4; the subsequent optimizations and code generation from HorseIR to C code in Sec. 5; Finally, we provide our experimental evaluation in Sec. 6, related work in Sec. 7, and conclusions in Sec. 8.

2 Background

2.1 Database Query Processing

SQL is based on relational algebra. RDBMS usually parse a SQL query into a relational algebraic representation [6], as the latter has been known to be easier to optimize [31]. Relational algebra operators are unary or binary, in the sense, they accept one or two tables[3] 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 operator tree and the input data.

Thus, modern RDBMS optimizers have a query re-write subsystem that generates multiple semantically equivalent relational algebra operator trees [15, 17] for a given query. 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 plan is then chosen. The cost model for this has traditionally focused on I/O cost as computations were assumed to be I/O bound [17].

However, as main memory has become cheaper and cheaper, RDBMS workloads have become compute bound, often not leveraging the underlying processor/memory architecture efficiently [3, 13]. This has resulted in the exploration of "block optimization" techniques [27, 36] to leverage CPU performance, with [4] being more sophisticated in their approach by accounting for CPU cache utilization.

2.2 Compilers Meet Databases

The importance of leveraging compiler optimization techniques for database query processing has gained traction in recent years [21]. A survey of compiler-based optimizations that have been attempted by the database community is presented in [34]. 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 well-known example is HyPer [26], an RDBMS which provides a compiler to translate SQL queries to LLVM directly (with some hybrid calls to C++ functions) before generating binary code. It is shown in [30], however, that introducing additional layers between SQL queries and low-level code can benefit from more compiler optimizations as higher-level intermediate languages have the potential to preserve more information from SQL queries.

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

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 applying an operation on all items of an array without an explicit loop. If an operation is mappable, it can be executed in parallel on each item of the array. For instance, MATLAB's elementwise built-in functions are well tuned for implicit data parallelism.

A special concept in 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 [9, 24].

HorseIR is influenced by ELI [8], Q [23], and Q'Nial [18]. It provides a set of special built-in functions, introduces homogeneous vector as a basic type, and proposes a set of list-based types 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 as this is needed to support database queries.

3 HorseIR: Design and Implementation

The design of HorseIR was motivated by the need for 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 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.

3.1 HorseIR: Language Design

At its core, HorseIR is a typed, 3-address intermediate representation with: a simple module system; static scoping; call-by-value semantics; a rich set of base types including a

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

wild-card type; key compound types including arrays, lists, dictionaries, tables, and keyed-tables; and a rich set of well-defined primitive array operations.

3.1.1 Program Structure

```
SELECT COUNT(*) AS StoresWithBigDiscount
FROM stores
WHERE discount >=0.5 AND discount <0.8;
```

```
module BigDiscount {
  import Builtin.*;  // import all builtins
  def main():table { // an entry method
// load table: stores
    a0:table = @load_table(`stores:sym);
// load column discount from table
    t1:f64 = check_cast(
             @column_value(a0,`discount:sym),f64);
// find all stores with discounts between [50%,80%)
    t2:bool = @geq(t1, 0.5:f64);
    t3:bool = @lt(t1, 0.8:f64);
    t4:bool = @and(t2, t3);
    // count the number of such stores
    t5:i64 = @sum(t4);
// return table
    t6:sym = `StoresWithBigDiscount:sym;
    t7:?   = @list(t5);  // ? => list<i64>
    t8:table = @table(t6, t7);
    return t8;
  }
}
```

Figure 2. An example HorseIR module (bottom) for an SQL query (top) which returns the number of stores with relatively big discounts (50-80%).

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, Fig. 2 shows an SQL query example and its corresponding example HorseIR module, `BigDiscount`. 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 `stores`, loads the column `discount`, and then executes the subsequent statements.

In addition to the named modules, there is also a predefined `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 [28] for modularizing complex software and provides a manageable way of specifying reusable libraries, such as the standard library `Builtin` in Fig. 2 which loads pre-defined functions, for example, load_table, geq, etc.

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 [12]. 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 a wild-card type.

3.1.2 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 allows for the case when a static type is unknown, thus indicating where a static type inference at compile time or a dynamic type check at runtime must be made. This tension between static and dynamic typing is partly due to the fact that database tables have declared types; thus, generating statically-typed HorseIR from queries should be possible, and is preferred. Furthermore, it has been well established that static types and shapes can lead to much more efficient array-based code, therefore one should aim for as much static typing as possible [22]. However, many common array languages are dynamically typed, and by offering a wild-card type, it is possible to generate HorseIR from those languages, thus providing the potential to use HorseIR for further purposes than pure DB queries, and opens the possibility of combining user-defined functions written in a traditional programming language [29] and SQL queries in a unified manner.

Base Types and Homogeneous Arrays A second key decision was to support quite a rich set of base types which includes: (1) all the base types typically found in an array-based language (boolean, char, short, int, long, float, double, and complex); (2) additional base types that are used in SQL (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, `t1:`**`f64`**, in Fig. 2 declares that `t1` 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. Furthermore, HorseIR supports many important built-in functions for dealing with these data structures, and they are used extensively in the code generation strategies described in Sec. 4.2.

A list type is a fundamental type which provides cells for holding different types and lists can be nested. Fundamental to list creation is the built-in function *list* which 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.

A specific subtype is associated with a list type when all cells of the list have the same subtype. For example, the variable `t7` in Fig. 2 is initially associated with a wild-card **?** but later inferred as a list type `list<i64>` at compile-time. It stores the value returned from the function `list` that has in this particular example only one input parameter, namely `t5`, which is an array of type **i64** (and in our example this array happens to be of size 1 as it was created by the `sum` primitive). Therefore, the type of the variable `t7`, `list<i64>`, means it is a list which has a set of cells (in our case only one), all of which are homogeneous arrays with a base type of **`i64`**. Finally, a table is formed with given column names (i.e. the variable `t6`) and column values (i.e. one column stored in the variable `t7`), where the number of column names is equal to the number of columns and all columns agree on their sizes.

Other advanced types derive from this nested list type such as dictionary, table, keyed table, and enumeration. A dictionary is a list of pairs of keys and values. A table is a special case of a dictionary. Each key represents a column name, and the associated value is an array representing the corresponding column of the table. Thus, the values of all pairs in the dictionary have the same length, namely the number of rows in the table. HorseIR supports the built-in functions *keys* and *values* for fetching keys and values from a dictionary. A keyed table is defined as a table with primary keys, which is distinct from a normal table. Since a keyed table may have a portion of columns as primary keys, our implementation contains two sub-tables for handling it: columns with primary keys are grouped as one sub-table and the other columns are grouped as another sub-table. This design simplifies our implementation by reusing the previous implementation of a normal table. An enumeration is used to maintain the mapping between two columns by storing the indices of the first occurrence of one column's value in the other one. Thus, it is introduced to emulate a key/value pair representing a foreign key in database tables. An example of enumeration is depicted in Fig. 3, and details of the use of the enumeration in HorseIR programs for SQL joins are described in Sec. 4.2.

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* loads a database table into a HorseIR table.

3.2 HorseIR: Language Implementation

We have implemented three core components for HorseIR: (1) a HorseIR front-end, which parses, performs semantic checks, and generates an AST from a HorseIR program; (2) a library of efficient and parallelized implementations for the rich set of built-ins for HorseIR; and (3) a compiler for HorseIR.

HorseIR Front-end The HorseIR front-end is built with Flex & Bison which are popular tools for building compilers. We defined a clean grammar for HorseIR. The front-end also performs the type and shape propagation, replacing wildcard 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 explicit 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. We use OpenMP to implement parallel C code with SIMD vectorization enabled in the library, and it is also convenient for generating efficient parallel code from fusing elementwise as operations described in Sec. 5. The design of HorseIR enables simple parallelization and exposes more information to subsequent optimizations since we support simple and clear combinations of vectors/arrays and lists.

However, parallelizing a single function is not sufficient because a synchronization barrier introduced after each operation is expensive. Thus, it is often beneficial to merge two or more functions by generating code on which loop fusion based optimizations have been applied. HorseIR exploits such fusion, generating specific fused C code for the set of fused statements with specific type and shape information.

HorseIR Compiler The HorseIR compiler uses a standard compiler design, which directly compiles the input HorseIR. Instead of generating specific C code for each built-in function, it generates C code with invocations to the pre-built functions in the built-in-function library, except when optimizations occur among statements that require specialized C code to be generated with specific type and shape information. In spite of the fact that the HorseIR compiler can generate efficient C code in terms of execution time because the code makes good use of the efficient built-in library functions, the cost of compiling the C code is quite expensive. In the future, we plan to add a JIT compiler with caching of compiled code in order to further reduce the overhead of compiling HorseIR and its compiled C code.

4 HorseIR Generation

In this section, we explain how to generate HorseIR programs from standard database SQL queries.

4.1 Starting from Optimized Execution Plans

Database systems have sophisticated query translators and optimizers that, given an SQL query as input, generate an optimized execution plan which is an operator tree, where its leaves are database tables and inner nodes represent relational operators: the most common ones being projection, selection, join, and various aggregation operators. By keeping statistics about data and by analyzing the selectivity of certain operators (the ratio of output vs. input records) the query optimizer decides on an order for the operators that typically aims at minimizing the number of records that flow from operator to operator, and that takes advantage of the best implementation of an operator depending on the characteristics of the data. Thus, it makes sense to take advantage of all these optimizations, and generate HorseIR code starting with an optimized execution plan. For this purpose, we use the query translator and optimizer of HyPer, a sophisticated column-based database system that exposes the execution plans it generates in the form of easily readable JSON objects. We take an optimized plan from HyPer and translate it into a HorseIR program. We would like to note that many database systems expose their execution plans in readable formats[5] and a similar translation to HorseIR is possible. In the following, we show how this translation is done.

4.2 Mapping Relational Algebra to HorseIR

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

Projection $\Pi_{a_1,\ldots,a_n}(R)$ takes the records of table R as input and returns the same records but only the columns of R with column names $a_1, \ldots, a_n$.[6] 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, $P = (P_1 <op> \ldots <op> P_n)$ where $<op>$ is either $\wedge$ for a logical AND operation or $\vee$ for a logical OR operation.[7] A predicate returns `True` or `False` when its input data satisfies a specific condition or not.[8] 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

[5] Database designers need them to decide on secondary index structures and database administrators need them to understand performance bottlenecks.

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

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

[8] 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.

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);
```

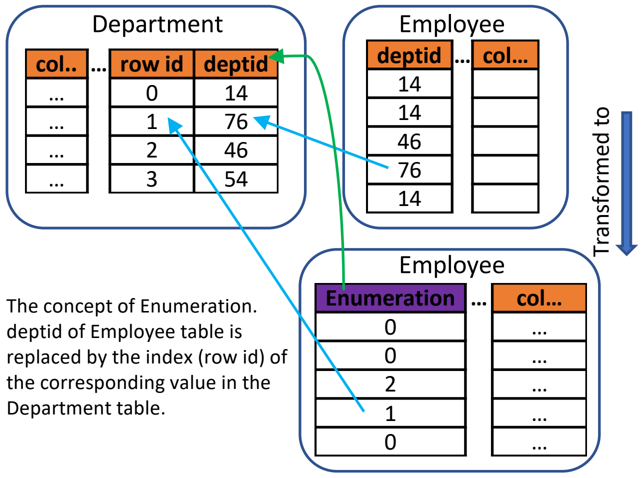
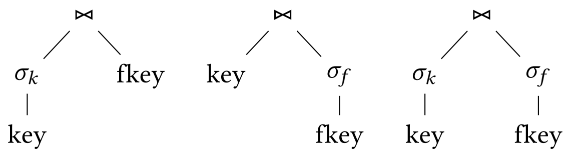

Department
col..
...
row id
deptid
0
14
1
76
2
46
3
54
Employee
deptid
col...
14
14
46
76
14
Transformed to
Employee
Enumeration
col...
0
0
2
1
0
The concept of Enumeration. deptid of Employee table is replaced by the index (row id) of the corresponding value in the Department table.

Figure 3. An example of an enumeration in a join with a pair of key and foreign key.



(a) Case 1 **(b)** Case 2 **(c)** Case 3

Figure 4. Three cases of joins on two columns (i.e. keyed and foreign keyed) when selections applied (i.e. σ_k and σ_f).

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 $COND$.

Since joining two tables is expensive, we provide a primitive data type, *enumeration*, that can be used if the join is over two tables with a primary/foreign key relationship. Each record of a table has a unique value in its *primary key* column, and each value in a *foreign key* column must be one of the values that exist in the primary key column of (usually another) a table. For example, in Fig. 3, `deptid` is a foreign key in `Employee` table referring to the primary key `deptid` of the `Department` table, establishing a primary/foreign key relationship between the two table. Joins over tables with a primary/foreign key relationship are extremely common as they link related tables. Thus, handling such joins efficiently is important. Therefore, we pre-compute such joins and store them as an enumeration.

An enumeration takes two parameters as a pair of <key (K), value (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. In our example, `deptid` of the `Employee` table is represented with the index (*row-id*) position of that value in the `deptid` column of the `Department` table. Enumerations are similar in principle to the *join index* concepts used in some RDBMS implementations [33]. The builtin function *enum* constructs an enumeration.

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

With enumeration, joins can be handled efficiently as follows:

1. In Fig. 4a, 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. In Fig. 4b, when a foreign key is selected with a filter, its key stays the same. We only need to update `ev` to `ev'` with fewer items in its fkey part.
3. In Fig. 4c, 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 case 2; (ii) update key to key'and fkey'to fkey''in case 1; and (iii) return a new enumeration with <key', fkey''>.

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 result, each cell of a list contains a single value. Finally, this is unraveled with the function *raze* and a vector is returned. The following example shows the aggregation function *count* on column G_1.

```
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);
```

4.3 Overall Code Generation Strategy

As we have just seen, each of the operators in an execution plan can be mapped into one or more lines of HorseIR code. An optimized plan (called plan in this section) has a nested tree-based structure (in the case of HyPer, it is in JSON format). Our code generator traverses the tree, generating code for each operator node. Code for each child is generated, and then the results passed to the operator.

Environment Objects In our code generation strategy, intermediate results are passed via an environment object. The object is similar to a table but has some additional elements for the purpose of delivering more information for optimizations at this translation level. Therefore, the object consists of:

- `table_name`: a string for a table name (a unique name is assigned for a temporary table);
- `cols_names`: a vector of strings for column names;
- `cols_alias`: a vector of strings for variables to represent the corresponding columns (it has the same length as `cols_names`);
- `cols_types`: a vector of strings for the corresponding variable types (it has the same length as `cols_alias`);
- `mask` : a string for masking (or no masking if its value is empty or null);
- `mask_a` : a vector of strings for variables to represent the corresponding columns after masking (it has the same length as `cols_alias`).

Expressions and Types Operators may have expressions as their children. Some expressions, such as `lt`, correspond directly to one HorseIR statement, whereas others, such as `between` require generating several HorseIR statements.

While generating HorseIR code, we take care to generate the most efficient types. For example, a `string` type in HyPer may be generated as a `symbol` type in HorseIR, if it corresponds to a read-only value. This results in more efficient code.

Eliminating Dead Code HorseIR code is generated assuming that all results will be needed. However, the final results table may contain only some columns, and thus we only need to retain the code needed to compute those columns. We use a backward slice from the final result to identify the code that needs to be retained, and eliminate all other code, as it is effectively dead code.

5 Optimizations and Code Generation

In this section, we introduce our optimizer which adopts a set of the loop fusion based optimizations and provides a brief description of emitting optimized C code.

5.1 Loop Fusion Based Optimizations

Loop fusion is a key optimization for array-based languages [10, 20], and it is similarly important for HorseIR. In our context, we have identified both fusions of elementwise functions, which is the normal case, as well as fusions/specializations based on patterns specific to the kind of HorseIR code generated for database operations.

Algorithm 1: Fuse elementwise functions

```
Data: HorseIR statements, UDChain, and DUChain
foreach stmt in statements with a reversed order do
    if isNotVisited(stmt) then
        FuseTree ← fuseElementwise(stmt);
        if number of nodes in FuseTree > 1 then
            /* Match and return FuseTree */
Function fuseElementwise(stmt):
    if isElem(fetchFuncName(stmt)) then
        if size(DUChain(stmt)) == 1 then
            rtn = [stmt]; setVisited(stmt);
            foreach def in UDChain(stmt) do
                rtn.append(fuseElementwise(def));
            return rtn;
    return ∅;
```

Fusing Elementwise Functions (FE). An elementwise function contains an implicit loop and no data dependency inside the loop. Therefore, elementwise functions are usually implemented with a parallel loop. It is ideal if two or more elementwise functions can be fused into a single loop when generating C code, to save the cost of synchronization between loops, to reduce loop overhead, and to provide larger loop bodies to the underlying C compiler optimizer.

We found that this sort of fusion is particularly useful for logic operations performed on boolean vectors (i.e. boolean short-circuit expressions). This is common in HorseIR code generated from the *WHERE* clause in SQL where predicates return boolean vectors for boolean operation `and`/`or`. Algo. 1 presents the algorithm for collecting those elementwise built-in function statements that can be fused. By leveraging UD and DU chains, a statement can trace all of its previous definitions and decide to fuse them together if the function of the statement is elementwise and the variable defined by the statement has only one use. The fusible elementwise statements are collected into a `FuseTree`, and then the code generation strategy will generate one fused loop for all of those statements, as discussed in Sec. 5.2.

Table 1. Fusing with Patterns: Code Examples

Name	Code Examples

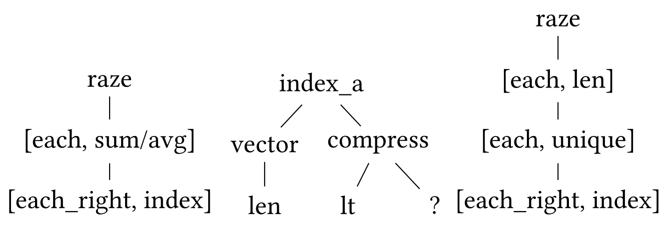
FP-1	`t0:? = @compress(mask, k0);`
	`t1:? = @compress(mask, k1);`

FP-2	`t0:? = @each_right(@index, k0, k1);`
	`t1:? = @each(@sum, t0);`
	`t2:? = @raze(t1);`

FP-3	`t0:? = @lt(k0, k1);`
	`t1:? = @compress(t0, k2);`
	`t2:? = @len(k2);`
	`t3:? = @vector(t2, 0:bool);`
	`t4:? = @index_a(t3, t1, 1:bool);`

FP-4	`t0:? = @each_right(@index, k0, k1);`
	`t1:? = @each(@unique, t0);`
	`t2:? = @each(@len, t1);`
	`t3:? = @raze(t2);`

Fusing with Patterns (FP). Fusing elementwise functions is beneficial, but there are other optimization opportunities for common patterns of array operations that occur in the code generated for the database operators. Therefore, a set of patterns, identified and adopted for optimizing these situations, can be found in Table 1. [9] Patterns are designed for merging statements and guiding our optimizer to generate efficient C code.



(a) FP-2 **(b)** FP-3 **(c)** FP-4

Figure 5. Patterns designed for FP-2, FP-3, and FP-4

Algo. 2 presents how a pattern can be matched in a HorseIR program. There are two directions for matching: (1) Top-to-Bottom needs the information of uses of statements by looking up DU chains (i.e. FP-1); and (2) Bottom-to-Top requires the information of definitions of statements by searching in UD chains (i.e. FP-2, FP-3, and FP-4), especially Fig. 5 shows that patterns are formed in a tree-based structure for Bottom-to-Top matchings. Once a pattern is matched, all matched statements need to be set visited and their information is saved for later code generation in Sec. 5.2.

Algorithm 2: Identify patterns for fusion

```
Data: HorseIR statements, UDChain, and DUChain
foreach stmt in statements do
    if stmt is assignment then
        if isTopToBottom then                              /* FP-1 */
            UseStmts ← DUChain(stmt);
              /* match if at least two statements in
            UseStmts contain the function compress
            and share the same mask */
        else                               /* FP-2, FP-3, and FP-4 */
            if fetchFuncName(stmt) == target then
                // 'raze' or 'index_a'
                if matchPattern(stmt, pattern) then
                    /* match!                                    */
Function matchPattern(stmt,pattern):
    value ← fetchValue(pattern);
    func ← fetchFuncName(stmt);
    if value == func then
        DefStmts ← UDChain(stmt);
        if size(DefStmts) ≠ size(pattern.child) then
            return False;
        for i = 0 to size(DefStmts)-1 do
            return matchPattern(DefStmts[i],
              pattern.child[i]);
        if pattern.name == "?" OR pattern.name == func then
            return True;
    return False;
```

```
t0:? = @gt(x, 10);    // x > 10
t1:? = @lt(x, 20);    // x < 20
t2:? = @and(t0, t1);  // t0 && t1
```

```
#define AND(x, y) (x) && (y)
#define GT (x, y) (x) > (y)
#define LT (x, y) (x) < (y)
for (i = ...){
  ... = AND(GT(x[i],10),LT(x[i],20));
}
```

Figure 6. An example code for fusing elementwise functions: HorseIR statements (top) and the generated C code (bottom).

5.2 Code Generation

A HorseIR program is compiled to C code. Instead of compiling to C code directly, we deploy multiple strategies for handling the following cases: (1) if a statement marked as visited is the root of a `FuseTree`, C code is generated by fusing statements with in-order traversal, for example, Fig. 6 shows that we provide a set of macros to assist the code generation of fused operations that makes the generated code clear; (2)

[9]The details of built-in functions and code examples can be found at: http://www.sable.mcgill.ca/~hanfeng.c/horse/docs/horseir/functions/

if a statement marked as visited matches a Top-to-Bottom pattern and it matches the last statement of the pattern, all statements matched are fused; (3) if a statement marked as visited matches a Bottom-to-Top pattern and it matches the root of the pattern, it is translated with a prepared template to optimized C code; (4) if a statement is only marked as visited, the statement is skipped (this operation will have been part of a `FuseTree`); and (5) if a statement is not marked as visited, it is translated to an invocation to a library function which is written in C (this statement was not found to be fusible with anything).

6 Evaluation

In this section, we present the results of our evaluation analyzing the overall performance of our approach, comparing against the performance of a state-of-the-art columnar RDBMS, MonetDB. Since MonetDB stores the columns of a table in consecutive space, its internal structure is considerably similar to an array. Furthermore, we provide a detailed analysis of the impact of the compiler optimizations deployed.

6.1 Methodology

Experimental Setup The experiments are conducted on a multi-socket multi-core server 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 C code with the optimization options `-O3` and `-march=native`; and MonetDB version v11.27.9. 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. We guarantee this for all systems 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. [10]

TPC-H SQL Benchmark Our tests use TPC-H [32], a widely used SQL benchmark suite for analytical data processing. TPC-H mimics a Business to Consumer (B2C) database application. It has 8 tables. The data is synthetically created and can vary in size. A *scale factor* (SF) of 1 corresponds to a database size of approximately 1 GB, with higher scale factors proportionately increasing the size of the database. The benchmark contains a suite of 22 SQL queries from simple to complex. Due to the limitations of our translator in terms of joins and the naive implementation of string matching for SQL `like` operation, we can handle so far of these queries (1, 4, 6, 12, 14, 16, 19, 22) and present results from them. Profiling data on the right side of Table 2 shows that these queries cover a variety of performance impacting dimensions such as the number of joins, condition predicates, and aggregations. For each of the queries, we took the execution plan generated by HyPer and translated it to HorseIR as outlined in Sec. 4, followed by compiler optimizations and code generation indicated in Sec. 5.

[10]https://github.com/Sable/dls18-analysis

6.2 Experimental Results

6.2.1 Overall Analysis

In a first experiment, we want to analyze the overall feasibility of our approach by analyzing the performance for scale factor SF1. We compare the performance of HorseIR without the compiler optimizations enabled (HorseIR-Nonopt), with compiler optimizations enabled (HorseIR-Opt) and with MonetDB. It should be noted that the time to pre-compute the enumeration data structures in HorseIR that represent the primary/foreign key relationship between the tables is not included in the execution time as the same structure can be used for many queries. Unfortunately, MonetDB does not provide its optimized execution plans in a readable format. A query is first translated into an unoptimized plan, which is then optimized and at the same time translated into an assembly-like language, called MAL, which makes plan analysis difficult. In particular, we do not know the details of their join implementations which might not have a pre-computed index available (as HorseIR does). Furthermore, MonetDB performs not only DB optimizations but also some compiler optimizations such as dead code elimination [25], while this is interweaved with code generation and not done at such a systematic level as in HorseIR. Thus, one has to be careful to draw detailed conclusions from the experiment. Instead, this experiment is intended to provide a first insight into performance.

Fig. 7 presents the execution times for the three approaches with increasing number of threads (from 1 to 64) for 8 of the queries. Overall, the execution times for MonetDB and HorseIR have, at large, the same order of magnitude. Given that MonetDB is a mature and considerably optimized database engine while HorseIR is still in a prototype status, these results are very promising showing that the use of array programming techniques for database query evaluation is worth investigating. In fact, for 6 of the 8 queries, HorseIR-Opt has actually lower response times than MonetDB, in one case (q16), it is slower, and in one case, both are close (q1). In contrast, HorseIR-Nonopt has the worst performance for 5 queries, lies in between HorseIR-Opt and MonetDB for two queries and is as good as HorseIR-Opt for one query (q22). This shows that the compiler optimizations are an important contribution to make an array-based implementation interesting. We discuss the impact of individual optimizations in a later experiment.

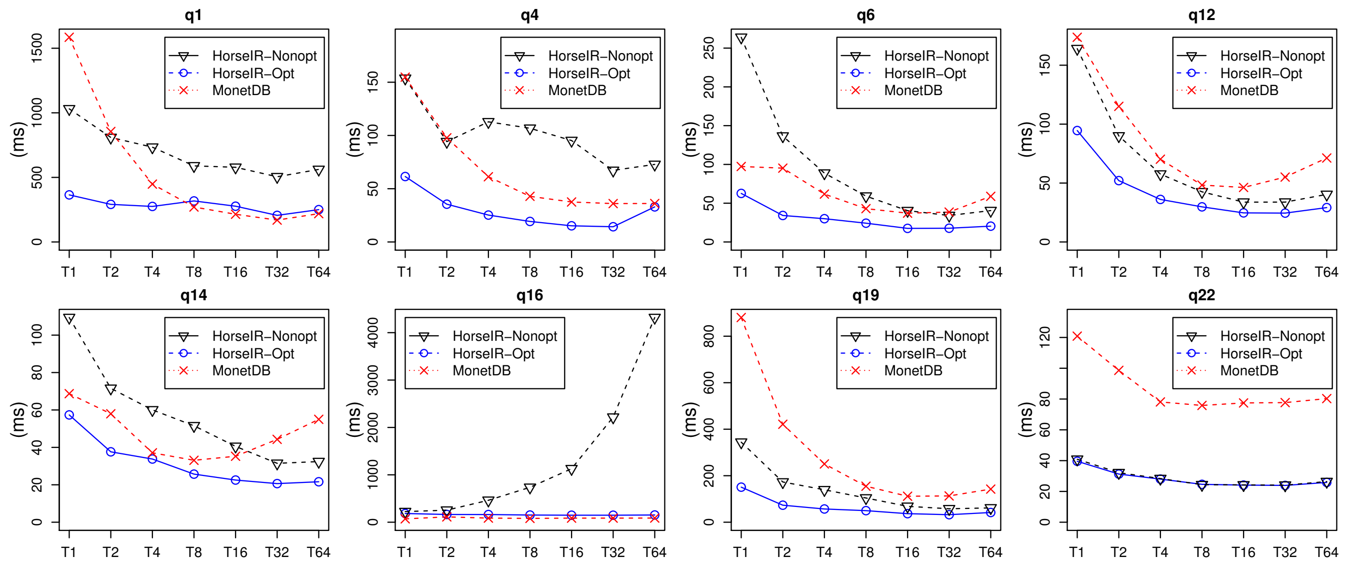


Figure 7. Performance comparison between HorseIR and MonetDB on SF1 with different number of threads, from 1 to 64.

In terms of parallelism, when the number of threads is increased, performance first improves but then remains stable or decreases. The improvement is most significant for MonetDB while HorseIR-Opt has already quite low execution times with few threads and then cannot improve as much with further parallelism. We assume the reason is the need to synchronize between threads.

An interesting case is q16 where HorseIR-Nonopt performs very bad with a large number of threads. The problem is the special shape of the input data for the function each, which is a long list with a short vector in each of its cells (e.g. 18314 cells with the average length of 6.5 vectors in SF1). Therefore, a cell-by-cell strategy is inefficient when the data in the cell is small. HorseIR-Opt fuses these statements (patter FP-4), avoiding this issue.

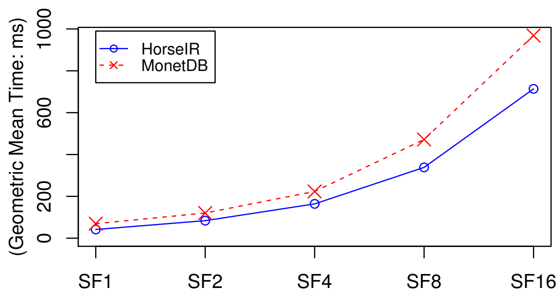


Figure 8. Geometric mean execution time for HorseIR and MonetDB with 16 threads (T16) across five different SFs.

As a second experiment, Fig. 8 shows the geometric mean over all queries for the optimized HorseIR and MonetDB with increasing scale factors when using 16 threads (which has the best performance for most queries). The behaviours of the two systems are very similar, showing excellent performance with a sublinear increase in execution times when data sizes increase. HorseIR performance overall better than MonetDB. This shows that our approach can easily handle larger data sizes without any further tuning.

6.2.2 Effect of Optimizations on HorseIR

Table 2. Query details left: number of optimization fusing elementwise functions (FE) and fusing with patterns (FP); right: number of joins, condition predicates, and aggregations.

Query	FE	FP-1	FP-2	FP-3	FP-4	Joins	Pred.	Aggr.
q1	2	1	2	0	0	0	1	8
q4	1	0	0	1	0	1	5	1
q6	1	1	0	0	0	0	4	0
q12	3	0	1	0	0	1	5	2
q14	4	1	0	0	0	1	3	1
q16	1	1	0	0	1	2	6	1
q19	6	2	0	0	0	2	21	1
q22	2	1	0	0	0	2	6	2

In order to study the effect of the different compiler optimizations, we tested queries compiled using HorseIR with four different optimization options: (1) No optimization; (2) Fusing elementwise functions only (FE-only); (3) Fusing with patterns only (FP-only); and (4) All optimizations (All-opt). The left side of Table 2 shows for each of the queries how often the different optimizations occur. To evaluate the impact of these optimizations under non-parallel and parallel environments we tested with 1 and with 16 threads.

Table 3 shows the speedup in execution time for the different optimized configurations compared to running with no

Table 3. Performance speedups on SF1 obtained by various HorseIR optimizations for different queries.

	FE-only		FP-only		All-opt	
Query	1 th.	16 th.	1 th.	16 th.	1 th.	16 th.
q1	1.10	1.05	2.25	2.08	2.83	2.09
q4	1.05	1.22	2.27	5.83	2.5	6.27
q6	3.19	1.58	1.09	1.35	4.22	2.30
q12	1.73	1.39	1.01	1.05	1.74	1.37
q14	1.22	1.11	1.43	1.50	1.91	1.80
q16	0.95	0.98	1.31	7.69	1.23	7.56
q19	1.47	0.97	1.31	1.65	2.28	1.86
q22	1.02	0.99	1.02	1.01	1.03	1.00
Geo. Mean	**1.35**	**1.14**	**1.39**	**2.06**	**2.03**	**2.39**

optimization enabled. A first observation is that the impact of these optimizations varies quite a bit among the different queries. For instance, for q1, FP has much more impact than FE, while the opposite is true for q6. Even though q6 has only one fusion for elementwise functions, it is the longest fusion chain, which fuses 9 statements. Some queries, however, barely benefit from any optimization, such as q22. Computing the geometric mean over all the queries, the speedup is around 2 for 1 thread, and 2.39 for 16 queries, which is considerable, meaning the queries run, on an average, in half the time with all optimizations enabled. For FE only, the number of threads has no impact on scale-up, while it does play a role for FP.

7 Related Work

Several approaches in the past have looked at exploiting compiler optimization techniques. A formal method for constructing a query compiler in Scala is introduced in [30]. This method emphasizes the use of multiple IRs to translate from SQL to C code so that optimization opportunities can be exposed on the different layers of IRs. HorseIR, on the other hand, uses a single IR.

HyPer aims in improving the query performance by compiling SQL to LLVM exploiting LLVM's compiler optimization mechanisms [26]. DBToaster targets high-performance delta processing in data streams by compiling SQL to C++ code [2]. These systems rely on a compiler for optimizing generated code, LLVM or C++. While these compilers are good at optimizing procedural code, they know little about what a query does at a conceptual high-level. HorseIR is able to optimize queries with a relatively high-level view by representing queries as arrays-based programs with less code but with more information about the queries themselves that it then exploits for compiler optimizations.

SciQL [35] provides array-based extensions to SQL. They use the array-based design to improve the performance of SQL queries and offer the possibility of mixing SQL queries and array programming. Compared to HorseIR, SciQL supports more general arrays targeted for scientific computing, while HorseIR keeps vector and list as primary data structures for efficient core SQL support.

HorseIR relies on many optimizations specifically developed for array programming languages to generate efficient parallel code, such as loop fusion known from array-based APL [10] and FORTRAN 90 [20]. However, focusing on code that represents SQL queries, these optimizations had to be adapted to the specific HorseIR context which goes well beyond a pure array programming language.

KDB+/Q [23], which was adopted in financial domains, attempts at fusing SQL and programming languages. Its database system was implemented in the array programming language Q, which is an interpreter-based language. It provides an SQL interface as a form of a wrapper on top of the language Q. The system internally maintains a database system, while seamlessly supporting an array programming language. However, with an interpreter-based design, its performance heavily relies on hand optimizations rather than systemic compiler optimizations.

8 Conclusion

In this paper, we discussed how the similarities between modern in-memory databases and array programming languages provide an opportunity for optimizing database query execution by leveraging compiler optimization techniques used in array programming languages.

We proposed HorseIR, an array-based intermediate representation that can be used to represent the SQL execution plans of an RDBMS. HorseIR code is generated automatically by a translator that takes as input the optimized SQL execution plan of an RDBMS (in our case HyPer). A variety of compiler optimization techniques are applied by the HorseIR compiler to generate efficient C code. Performance results using the SQL TPC-H Benchmark testify that our implementation of HorseIR is on par with current state-of-the-art column-based RDBMS, and show that applying compiler optimization techniques over RDBMS execution plans provides substantial performance benefits. Further, the results also demonstrate that the multi-threading capability 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 explore new optimizations for array-based primitive functions integrating with database heuristics, in order to enable 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. Furthermore, we also plan to explore the possibility of performing JIT compilation of HorseIR with support for both CPUs and GPUs.

References

[1] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Special Interest Group on Management of Data (SIGMOD)*. ACM, 967–980.

[2] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *The Proceedings of the VLDB Endowment (PVLDB)* 2 (2009), 1566–1569.

[3] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. 1999. DBMSs On A Modern Processor: Where Does Time Go?. In *Conference on Very Large Data Bases (VLDB)*. 266–277.

[4] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research (CIDR)*. 225–237.

[5] Qiang Cao, Pedro Trancoso, J-L Larriba-Pey, Josep Torrellas, Robert Knighten, and Youjip Won. 1999. Detailed Characterization of a Quad Pentium Pro Server Running TPC-D. In *International Conference on Computer Design (ICCD)*. IEEE, 108–115.

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

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

[8] Hanfeng Chen and Wai-Mee Ching. 2013. ELI: A Simple System for Array Programming. *Vector, the Journal of the British APL Association* 26, 1 (2013), 94–103.

[9] Hanfeng Chen, Alexander Krolik, Erick Lavoie, and Laurie J. Hendren. 2016. Automatic Vectorization for MATLAB. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*. 171–187.

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

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

[12] Vincent Foley-Bourgon and Laurie J. Hendren. 2016. Efficiently Implementing the Copy Semantics of MATLAB’s Arrays in JavaScript. In *Dynamic Languages Symposium (DLS)*. 72–83.

[13] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *Special Interest Group on Management of Data (SIGMOD)*. ACM, 981–992.

[14] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

[15] Yannis E Ioannidis. 1996. Query Optimization. *Comput. Surveys* 28, 1 (1996), 121–123.

[16] ISO/IEC 9075-1:2016 2016. *Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. Standard. International Organization for Standardization.

[17] Matthias Jarke and Jurgen Koch. 1984. Query Optimization in Database Systems. *Comput. Surveys* 16, 2 (1984), 111–152.

[18] Michael A. Jenkins. 1989. Q’Nial; A Portable Interpreter for the Nested Interactive Array Language, Nial. *Software: Practice and Experience* 19, 2 (1989), 111–126.

[19] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *International Conference on Data Engineering (ICDE)*. 195–206.

[20] Ken Kennedy. 2001. Fast Greedy Weighted Fusion. *International Journal of Parallel Programming* 29, 5 (2001), 463–491.

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

[22] Vineet Kumar and Laurie J. Hendren. 2014. MIX10: compiling MATLAB to X10 for high performance. In *Conference on Object-oriented Programming, Systems, and Applications*. 617–636.

[23] kx. 2018. KDB+/Q Database System. Retrieved June 2018 from https://kx.com/

[24] Vijay Menon and Keshav Pingali. 1999. A Case for Source-level Transformations in MATLAB. In *Conference on Domain-specific Languages (DSL)*. ACM, 53–65.

[25] MonetDB. 2018. MonetDB Optimizer Pipelines. Retrieved June 2018 from https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/OptimizerPipelines

[26] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *The Proceedings of the VLDB Endowment (PVLDB)* 4, 9 (2011), 539–550.

[27] Sriram Padmanabhan, Timothy Malkemus, Anant Jhingran, and Ramesh Agarwal. 2001. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *International Conference on Data Engineering (ICDE)*. IEEE, 567–574.

[28] David Lorge Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[29] Mark Raasveldt and Hannes Mühleisen. 2016. 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*. 16:1–16:12.

[30] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Special Interest Group on Management of Data (SIGMOD)*. 1907–1922.

[31] John Miles Smith and Philip Yen-Tang Chang. 1975. Optimizing the Performance of a Relational Algebra Database Interface. *Commun. ACM* 18, 10 (1975), 568–579.

[32] Transaction Processing Performance Council. 2017. TPC Benchmark H.

[33] Patrick Valduriez. 1987. Join Indices. *ACM Transactions on Database Systems (TODS)* 12, 2 (1987), 218–246.

[34] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Trans. on Knowl. and Data Eng.* 27, 7 (2015), 1920–1948.

[35] Ying Zhang, Martin L. Kersten, and Stefan Manegold. 2013. SciQL: Array Data Processing Inside an RDBMS. In *Special Interest Group on Management of Data (SIGMOD)*. 1049–1052.

[36] Jingren Zhou and Kenneth A Ross. 2004. Buffering Databse Operations for Enhanced Instruction Cache Performance. In *Special Interest Group on Management of Data (SIGMOD)*. 191–202.