McGill University
School of Computer Science
Sable Research Group

Accelerating Database Queries for Advanced Data Analytics: A New Approach

Sable Technical Report No. SABLE-TR-2021-1

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

Feburary 15, 2021

Contents

List of Figures

List of Tables

Abstract

The rising popularity of data science in recent times has resulted in the diversification of data processing systems. The current ecosystem of data processing software consists of conventional database implementations, traditional numerical computational systems, and more recent efforts that build a hybrid of these two systems. As many organizations are building complex applications that integrate all the three types of data processing systems, there is a need to look at a holistic optimization strategy that can work with any of the three, or their combinations. In this paper, we propose an advanced analytical system HorsePower, based on HorseIR, an array-based intermediate representation (IR). The system is designed for the translation of conventional database queries, statistical languages, as well as the mix of these two into a common IR, allowing to combine query optimization and compiler optimization techniques at an intermediate level of abstraction. Our experiments compare HorsePower with the column-based database system MonetDB and the array programming language MATLAB, and show that we can achieve significant speedups for standard SQL queries, for analytical functions written in MATLAB and for advanced data analytics combining queries and UDFs. The results show a promising new direction for integrating advanced data analytics into database systems by using a holistic compilation approach and exploiting a wide range of compiler optimization techniques.

Accelerating Database Queries for Advanced Data Analytics: A New Approach

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

Feburary 15, 2021

1 Introduction

Complex data analytics has become the cornerstone of our data-driven society. Although the amount of data stored in traditional relational database systems (DBS) has been growing rapidly, the by far most common current approach is to take the data first out of the DBS and load it into stand-alone analytical tools, which are often integrated programming language systems such as Python, MATLAB [1], and R [3]. When needed, the results of such a DBS external analysis can be reintegrated into the database. However, as the size of the data increases, the expensive data movement between DBS and analytics tools can become a severe bottleneck.

An alternative that avoids such data movement is to integrate the analytical capabilities into the DBS. The most well-known approach to do so is to support user-defined functions (UDFs) written using a conventional, high-level programming language, that are then embedded in SQL queries [29, 24, 36, 33]. For example, MonetDB [29] supports the integration of a Python interpreter into its DBS execution environment, allowing users to include Python functions inside their SQL queries. These functions are then executed by a language interpreter (Python) that is embedded inside the DBS engine. AIDA [10] takes this a step further by offering developers a more intuitive interaction between Python code and SQL statements, but internally leverages the UDF constructs offered by the back-end DBS.

Although UDF implementations integrate SQL queries with high-level programming language functions, they still have separate execution environments: one being the SQL execution engine, and the other the programming language execution environment. This can quickly lead to costly data format conversion between the two environments. Furthermore, there is a clear delineation between the declarative query part, written in SQL, and the UDF, written in a higher-level programming language, and both parts are individually optimized by their respective execution environments, without the consideration of any holistic optimization across the entire task.

In this paper, we address the shortcomings of such separated execution environments. In particular, we propose `HorsePower`, an advanced analytical SQL system, which provides a holistic solution for seamlessly integrating analytical functions into SQL queries. As depicted in Figure 1, the system is based on HorseIR [7], an array-based intermediate representation (IR) language which was developed to explore the usage of compiler optimizations for query execution. Chen et al. [7] translated the execution plans of standard SQL queries into HorseIR and compiled the generated HorseIR code using various compiler optimization strategies developed for array-based languages. The resulting low-level target code is then executed on the data. Using arrays to represent database columns,

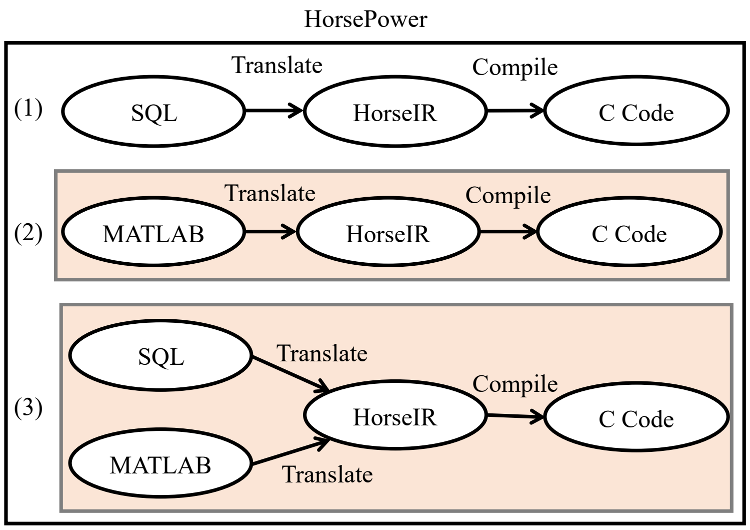

Figure 1: HorsePower overview: execution environment for SQL queries, MATLAB functions and MATLAB UDFs embedded in SQL code using the array-based intermediate representation language HorseIR

HorseIR follows conceptually the data model of column-based DBS, which has been proven to be effective for data analytics tasks [4, 14].

In this paper, we present HorsePower, which extends the idea to a full-fledged execution environment for data analytics. Additionally to supporting plain SQL queries, HorsePower also supports pure MATLAB functions as well as their usage as UDFs that can be embedded in SQL queries. MATLAB is a popular high-level array language widely used in the field of statistics and engineering. HorsePower can take analytical functions implemented in MATLAB and translate them to HorseIR code, to be optimized and executed as independent executables. Furthermore, these functions can also be embedded in SQL in the form of UDFs. In this case, both SQL and MATLAB code are transformed and merged into a single HorseIR code base before being optimized in a holistic manner.

As such HorsePower avoids the overhead of inter-system data movements as it has a single execution environment, and eliminates the barriers between SQL queries and analytical functions allowing to optimize across both the declarative and functional parts of the query.

In summary, the contributions of this paper are as follows:

- We present HorsePower, and advanced analytical system, that extends the approach proposed in [7] to not only offer a compiler-based execution environment for SQL queries, but also for programs written in the array-based language MATLAB and for SQL queries with embedded UDFs.
- HorsePower uses a holistic approach of exploiting array-based compiler optimization techniques for both SQL and MATLAB taking advantage of the conceptual similarities of columns and arrays.
- The performance of HorsePower is shown through an extensive set of experiments on programs written in MATLAB, and SQL queries with embedded UDFs.

In the rest of the paper, we first introduce background in Section 2; present HorsePower in Section 3;

evaluate the performance of benchmarks in Section 4; discuss related works in Section 5; and conclude in Section 6.

2 Background

In this section we introduce the research background in database query compilers, HorseIR as an intermediate representation, and traditional UDFs.

2.1 Database Query Compilers

An SQL query can be processed using two different execution models, the Volcano iterator model [13] and the data-centric model [25]. The volcano model represents the classical approach which fetches tuples through a set of chained operations in a pipelined fashion. This avoids large intermediate results as the tuples are produced as needed. In contrast, the data-centric model aims in merging chained operations avoiding intermediate results when possible.

The data-centric model has gained recent attention with the development of modern query compilers that translate an SQL query into an intermediate representation (IR) before target code is generated from the IR. This approach reduces the engineering difficulty that is associated with generating binary code directly from the SQL query while also making it possible to leverage any existing code optimizations available within the IR platform. Although a direct comparison is absent, such data-centric approaches promise to deliver better performance [31].

HyPer [25] is a well-known DBS using the data-centric model. HyPer compiles SQL execution plans to LLVM, a low-level IR for imperative programming languages[1], and the code optimizations within the existing LLVM compiler infrastructure are exploited. HorseIR, on which HorsePower is based, is an array-based IR, making it particularly interesting for column-based query optimizations. MonetDB translates execution plans to its own intermediate code, called MAL. However, the code is then run in interpreted mode and as such does not leverage any compiler optimizations that are possible with binary code.

So far, all these systems are mainly focused on SQL queries without UDFs, not considering a holistic analysis of both analytical functions and SQL in their optimization pipelines.

2.2 HorseIR: an Array-based IR for SQL

HorseIR [7] was developed as a high-level IR specifically for database applications, and its compiler follows the data-centric model to generate target code [8]. Being an array-based IR, it is relatively straightforward to generate basic HorseIR code following the execution plans developed by column-based DBS, as the operators executing on entire columns can be translated to functions executing on vectors in HorseIR. In fact, Chen et al. [7] took HyPer's execution plans, that incorporate a wide range of traditional DBS optimizations, as the input for generating HorseIR programs.

In this regard, HorseIR provides a rich set of array-based built-in functions to which one can map the standard database operations. Moreover, the HorseIR compiler provides vital optimizations over these array-based operations. For example, *loop fusion* merges multiple loops into one loop,

[1] https://en.wikipedia.org/wiki/Imperative_programming

```sql
SELECT
  SUM(l_extendedprice * l_discount) AS RevenueChange
FROM
  lineitem
WHERE
  l_discount >= 0.05;
```

(a) Example query derived from the TPC-H benchmark.

```
 1 module ExampleQuery{
 2   def main(): table{
 3     // load table
 4     t0:table = @load_table(`lineitem:sym);
 5     // load two columns
 6     t1:f64 = check_cast(@column_value(
 7         t0, `l_extendedprice:sym), f64);
 8     t2:f64 = check_cast(@column_value(
 9         t0, `l_discount:sym), f64);
10     // compute revenue change
11     t3:bool = @geq(t2, 0.05);
12     t4:f64  = @compress(t3, t1);
13     t5:f64  = @compress(t3, t2);
14     t6:f64  = @mul(t4, t5);
15     t7:f64  = @sum(t6);
16     t8:sym  = `RevenueChange:sym;
17     t9:list<f64> = @list(t7);
18     t10:table = @table(t8, t9);
19     return t10;
20   }
21 }
```

(b) Corresponding HorseIR code

Figure 2: Example query and its HorseIR program

allowing for an intuitive merge of chained operations and thus, avoiding intermediate results. Thus, optimizations developed for array-based programming languages can be exploited to improve query performance.

HorseIR Example Figure 2a shows a simplified version of Query 6 of the TPC-H benchmark [35] computing the change in total revenue given prices and discounts from the table `lineitem` for all those items where the discount is at least 0.05. A basic translation of this query into a HorseIR program (prior to performing any optimizations) is shown in Figure 2b. In the HorseIR program, the function `main` defines the entry of the program and returns a table as the final result. It first obtains the reference to the table `lineitem`[2] and then uses it to obtain the references to the data sets of the columns `l_extendedprice` and `l_discount` (both being vectors) in variables `t1` and `t2` respectively. Then, the predicate is evaluated by invoking the built-in function `@geq`. This function returns a boolean vector of the same length as `t2` with a `true` value for each row where the corresponding `t2` row has a value of at least 0.05, and a `false` otherwise. Next, the function `@compress` in lines 12 and 13 takes this boolean vector and `t1` resp. `t2` as input, and returns all rows from `t1` resp. `t2` where the corresponding values in the boolean vector are `true`. The length of the resulting vectors `t4` and `t5` is equal to the number of true values in the boolean vector `t3`. Finally, the multiplication function in the `SELECT` clause is executed on the two vectors in lines 14, and the summation in line 15. In the end, in lines 16 to 19, a new table, with a single column named `RevenueChange` and a single row for

[2]HorseIR is an in-memory system, where all the tables are primarily memory-resident.

```
...
revenue = 0;
for(i = 0; i < numRows; i++){
    if(t2[i] >= 0.05)
        revenue += t1[i] * t2[i];
}
...
```

Figure 3: Optimized C code for the IR code in Figure 2b

the total revenue change, is created and returned.

HorseIR Optimizations The execution steps in the above program are conceptually similar to those executed by MonetDB over its MAL code. As can be seen, such an approach generates a fair amount of intermediate results. In particular, `t3` to `t6` are all intermediate vectors that are materialized. If lines 11 to 15 are translated to lower-level code independently, each of them generates its own `for` loop over the corresponding arrays. However, array-based optimization techniques, including loop fusion, and some pattern-based optimizations developed specifically for the operator sequences found in SQL statements, allow the HorseIR compiler to fuse these loops to just one loop to avoid materializing these intermediate vectors. The resulting sequential C code after such optimizations is similar to the one depicted in Figure 3. The various optimization techniques will be discussed in-depth in Section 3.4. Although the source code in Figure 3 does not convey it explicitly, behind the scenes, HorseIR uses OpenMP to compile them into a parallel implementation, as outlined in [7].

2.3 Traditional Database UDFs

A UDF is a high-level language function embedded within an SQL statement. It can simplify a query by offloading its partial computation in a more concise language other than SQL, or it can provide additional functionality that cannot be expressed by SQL alone. Some DBS offer their own vendor-specific SQL language extensions to write UDFs, such as Transact-SQL UDFs in Microsoft SQL Server [30]. Moreover, many DBS provide interfaces for integrating general-purpose programming languages into SQL queries, such as the embedded Python interpreter in MonetDB [29]. Using well-known programming languages for UDFs has become a popular choice as it simplifies software development.

UDFs are often classified into subcategories depending on their expected interaction with the SQL query. For the sake of brevity, we will focus only on *Scalar UDFs* and *Table UDFs*, as these are the most commonly employed types of UDFs and also the ones supported presently in HorsePower. An avid reader can refer to [29] for a detailed discussion on the various UDF categories. To better understand UDFs and how they are executed in traditional DBS, Figure 4 shows rewrites of the query of Figure 2a where part of the calculation is outsourced to a UDF, in (a) to a scalar UDF, and in (b) to a table UDF.

Scalar UDF A *scalar UDF* returns a single value per row (which could be a vector) and can be therefore essentially used wherever a regular table column is used, such as the `SELECT` or the `WHERE` clause of SQL queries. Figure 4a shows a scalar UDF which performs the multiplication that was originally part of the `SELECT` clause in Figure 2a. Although this is a simple example, outsourcing such computations to a UDF extends its use across several queries, and allows for a simpler change of the implementation or semantics of the UDF (and therefore, that of the queries using it).

```
FUNCTION calcRevenueChangeScalar(price,discount)
  RETURN price * discount;
END
```

```
SELECT
  SUM(calcRevenueChangeScalar(l_extendedprice,l_discount)) AS RevenueChange
FROM
  lineitem
WHERE
  l_discount >= 0.05;
```

(a) Example query with a `scalar` UDF

```
FUNCTION calcRevenueChangeTable(price,discount)
  revenuechange = price * discount;
  RETURN TABLE("revenuechange", revenuechange);
END
```

```
SELECT
  SUM(revenuechange) AS RevenueChange
FROM
  calcRevenueChangeTable((
    SELECT l_extendedprice, l_discount
    FROM lineitem
    WHERE l_discount >= 0.05));
```

(b) Example query with a `table` UDF

Figure 4: Rewriting the example query with UDFs

In a row-based system, this query retrieves one tuple after the other from the `lineitem` table, and if the condition in the `WHERE` clause is true, the values in the `l_discount` and `l_extendedprice` attributes are given to the UDF, which performs the multiplication. Thus, the UDF is executed for each row that fulfills the `WHERE` clause.

In contrast, in a column-based system, and under the assumption that the programming language used for the UDF can handle array-based data structures, the execution is quite different. Let's look at the execution within MonetDB. As we have already indicated, MonetDB executes similar in spirit to our unoptimized HorseIR program depicted in Figure 2. First, the `WHERE` clause is executed on the `l_discount` column of all the rows, returning a boolean vector of the same size as `l_discount` with true values in the elements (rows) that fulfill the condition. Then, MonetDB applies the corresponding boolean selection on columns `l_discount` and `l_extendedprice` resulting in "compressed" columns only containing the elements of `l_discount` and `l_extendedprice` for which the corresponding entry in the boolean vector was true. These two compressed columns are then given to the UDF as arrays which perform an elementwise multiplication on these arrays returning an array of the same size. This is then the input to the `SUM` operator. Thus, compared to a row-based system, the UDF is only called a single time and works on arrays instead of individual values.

Table UDF A *table UDF* returns a table-like data structure, and thus, is typically called within the `FROM` clause of an SQL statement, similar to regular database tables. As such, it can return one or more columns at the end of its execution. Figure 4b shows a table UDF which is specifically designed for a column-based system. The input for the table UDF is the `l_extendedprice` and `l_discount` columns of the `lineitem` table. Similar to the scalar UDF example, the database performs the selection operation based on the value of the `l_discount` column. It then passes the values of the `l_extendedprice` and `l_discount` columns for the selected rows to the table UDF. As such, each input column to the table UDF is a vector of values for the corresponding columns, equal in length to the number of selected records from the `lineitem` table. The table UDF then performs the element-wise multiplication function, and returns a table-like data structure with the result of this multiplication as the column `revenuechange`. This resulting table-like data structure is then the input of the surrounding SQL query, which uses it in its `FROM` clause and executes the `SUM` operation on `revenuechange`.

Discussion The main advantage of UDFs is that the core computation of the query is abstracted into a function. Whether to use a scalar or a table UDF depends on the expertise of the developer and task to be performed. For developers, it is often unclear which one is more performant, and they might simply prefer one style because they find it more intuitive. Furthermore, they have quite different interfaces. Thus, some problems might be easier expressible as table UDF or as scalar UDF (or might only be expressed by one of these types). In the case of column-based systems and the use of programming languages that support operators on arrays, such as R, MATLAB or Python, column-based data can be exchanged seamlessly between SQL and the embedded UDF interpreter, and efficient array operations can be exploited within the UDF.

However, introducing UDFs into queries can bring performance issues. Firstly, the overhead of possible data movement and conversion is non-negligible when data materialization is required for the input of a UDF and as well when the return value of a UDF is given back to the SQL statement. This is because often the data types used by the two execution environments are not similar, requiring the implementation to perform some data conversion from one format to the other. In fact, MonetDB tries to avoid this by employing a concept called *zero-copy* [21], whereby a high-level language UDF can directly access the DBS buffers of a column if the underlying binary structures of the data types used in both the UDF and the DBS are very similar.

Apart from the data format issue, the fact that there is a large syntax gap between SQL and the UDF language results in both parts of the query being executed as black boxes to each other by two separate execution environments. In other words, there is typically no cross optimization between the two components of the query. An exception is the approach proposed in Froid [30], where UDF code is rewritten to SQL code so that the query optimizer of the database system can optimize across the entire rewritten query. This is possible with the examples that we discussed above, as the tasks performed by the UDFs can be expressed as SQL operators. However, this is not the case for all the tasks for which UDFs are currently used, such as those involving non-relational operations on the data.

Hence, we propose a new solution where both the UDF and the SQL components of a query are translated into a single HorseIR program. This allows for a holistic optimization of the entire query and also avoids any data movement or conversion.

3 HorsePower

In this section we present the design and implementation of HorsePower. As depicted in Figure 1, it is a system which is designed for the code generation and optimization of HorseIR generated from (1) SQL queries, (2) MATLAB programs, and (3) SQL queries with analytical functions written in MATLAB. We first discuss, how HorsePower translates each of these three program types to HorseIR code. Then we describe how the resulting HorseIR programs are optimized and compiled to target C code for execution.

3.1 SQL to HorseIR

The translation from SQL to HorseIR presented in the prior work [7] uses HyPer to generate optimized execution plans which are then translated to HorseIR. These execution plans are expressed as JSON objects, which made for a relatively direct translation to HorseIR.

However, the HyPer interface does not support UDFs. Therefore, HorsePower is instead based on MonetDB's execution plans as MonetDB supports UDFs and its execution plans contain the relevant UDF information such as function names, parameters and parameter types. As MonetDB's plans follow a traditional tree structure, HorsePower transforms this tree structure to a JSON object, which can then be translated to HorseIR with the infrastructure presented in [7] with some extensions to handle UDF information. For a plan that contains a UDF, we describe additional steps in detail in Section 3.3[3].

Previous work on HorseIR [7] only supported a subset of SQL and did not properly support multi-join queries. The current version of HorsePower has been significantly extended and is now able to execute all queries of the TPC-H benchmark [35]. As these extensions relate to traditional database operators rather than aspects of compiler optimization, we do not present the details here.

3.2 MATLAB to HorseIR

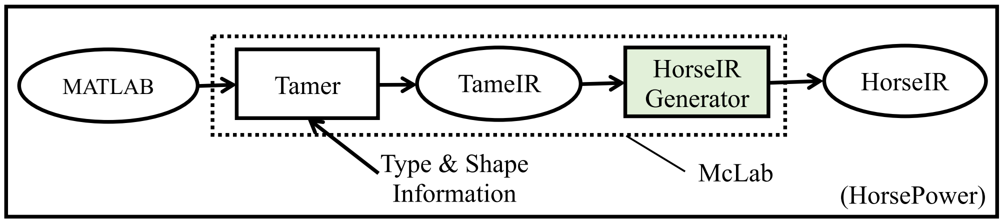

Figure 5: Generating HorseIR code from MATLAB using the McLab framework in HorsePower

We choose MATLAB as the source array language to be supported by HorsePower because it is a very popular language in the fields of statistics and engineering. As such, we have to provide a MATLAB-to-HorseIR translator for transforming MATLAB code into HorseIR. Since MATLAB is a sophisticated dynamic language which provides numerous flexible language features, it is challenging to build a MATLAB compiler from scratch. Thus, we employ the McLab framework [2] which provides a complete solution for parsing, analyzing, and optimizing MATLAB programs, and generating target code. Figure 5 shows the full workflow for compiling MATLAB to HorseIR. The McLab framework translates MATLAB programs to its own internal IR, called TameIR, which was specifically designed to enable optimization of MATLAB programs. From there, TameIR can be translated into various other programming languages to build efficient executable code [22, 19]. Thus we extend the McLab framework to include a HorseIR generator that can translate TameIR to HorseIR code.

[3]In principle, HorsePower could have its own ”homemade” query compiler to general execution plans. However, as the traditional relational query optimization techniques are not the focus of our research, we preferred to integrate the already optimized execution plans generated by existing DBS into our prototype.

The translation from MATLAB to TameIR, performed by the Tamer module, has to handle MATLAB's many dynamic features and the lack of strict typing. When analyzing the program, the first set of type and shape information is derived from the parameters of the entry MATLAB function. This information is then used to derive the type and shape information for any further variables computed by the statements in the rest of the program. From there, classic program analysis and optimizations are performed, including interprocedural value analysis, constant propagation and common subexpression elimination to produce as output optimized TameIR code [11]. TameIR can represent MATLAB's matrix and high-dimension arrays, and currently supports an essential subset of MATLAB array operations, such as matrix multiplication and inverse.

However, compared to HorseIR, TameIR lacks support for advanced types that are needed for query executions, such as the `table` data type that is essential for organizing a collection of columns. Furthermore, it does not provide the database-related functions that HorseIR supports, and the data-centric optimizations incorporated into the HorseIR framework. To bridge this gap, HorsePower translates the generated TameIR programs into HorseIR.

So far, this translator supports a core subset of MATLAB features and built-in functions as follows.

Functions. When compiling multiple MATLAB files, the first function is considered an entry function, i.e. the main function. Since both MATLAB and HorseIR support the pass-by-value parameter passing, the code generation is straightforward for parameters. However, HorseIR provides an optimization over the default pass-by-value approach by internally employing a copy-on-write mechanism. In this approach, if it is determined that the function does not modify a parameter that is passed to it, then such a parameter is provided through pass-by-reference, saving any overhead associated with making data copies.

Control structures. The common control structures `if-else` and `while` are supported by both languages. When testing a condition in a control structure, the result must be a single boolean element which can be a one-element vector. While in MATLAB a condition check is also considered `true` when a conditional expression evaluates to a non-empty set of elements, this is disallowed for the program translation to HorseIR as this is currently unsupported in HorseIR.

Arrays. We support MATLAB programs in an array programming style without using the `for-loop` construct for loop iteration in MATLAB. Instead, programs can use the MATLAB built-in functions which can operate on whole vectors. For example, array indexing is an important operation for data materialization of selected rows by given row numbers. More precisely, given the MATLAB code for array indexing `A(I)`, if `I` is an array of boolean values of the same length as `A`, then this is called logical indexing and equivalent to the function `@compress` in HorseIR that we have already discussed. If $I = (i_0, i_1, ...)$ is a vector of integer indices, it can be represented with the built-in function `@index` in HorseIR, and results in a new vector $(A[i_0], A[i_1], ...)$ whose length is equal to that of `I`.

Types. HorseIR has support for a rich set of types, some of which can be directly mapped to MATLAB types supported in the McLab framework, including boolean, character, integer, and floating point. For example, the floating point value `double` in McLab is translated to `f64` in HorseIR.

Shapes. Obviously, we support MATLAB arrays as they are essential components for HorseIR to work on table columns. An array in MATLAB can be either a `1-by-N` or `N-by-1` matrix where

```
module ExampleQuery{
  def calcRevenueChangeScalar(price:f64, discount:f64): f64{
     x0:f64 = @mul(price, discount); // S5
     return x0;
  }
  def main(): table{
     ...
     // compute revenue change
     t3:bool= @geq(t2, 0.05:f64);     // S0
     t4:f64 = @compress(t3, t1);      // S1
     t5:f64 = @compress(t3, t2);      // S2
     t6:f64 = @calcRevenueChangeScalar(t4,t5); // S3
     t7:f64 = @sum(t6);               // S4
     ...
}}
```

Figure 6: HorseIR code for the UDF in Figure 4a

N is a positive integer greater than 1. We support the `1-by-N` vector as its data layout is more cache-friendly in MATLAB.

3.3 SQL and UDF to HorseIR

HorsePower supports SQL queries with embedded UDFs written in MATLAB. As described in Section 3.1, HorsePower uses the execution plans generated by MonetDB. While MATLAB is currently not in the list of languages supported by MonetDB for its UDF implementations, this is irrelevant for generating the execution plans as only the hooks into the UDFs with their input and output parameters are relevant. This is because MonetDB treats all UDF execution environments as a black-box and generates the same execution plan, independent of the language of the UDF implementation. Thus, we can directly use MonetDB's execution plan generator on SQL statements extended with UDFs. From there, we use the plan-to-HorseIR translator introduced in Section 3.1 to first generate HorseIR code from the plan with placeholders for UDF method invocations. That is, in the HorseIR code, the invocation of the UDF is simply translated into a method invocation in HorseIR. Next, we generate a separate piece of HorseIR code by translating the UDF written in MATLAB using the MATLAB-to-HorseIR translator introduced in Section 3.2. Finally, the two segments of code for SQL and UDFs are integrated into a single HorseIR program.

As discussed in Section 2.3, our current implementation supports two kinds of UDFs, namely scalar and table UDFs. In order to make the MATLAB functions to conform to the semantic form expected of these types of UDFs, we enforce the following restrictions on the implementation of such MATLAB functions. (1) A function must have one return statement with either a single vector (for scalar UDFs) or a table-like data structure (for table UDFs). (2) Executing the MATLAB function individually on each scalar element of an array and returning the result set as an array should be equivalent to executing the function on the full array as input (that is, f($\{x_1,x_2,..,x_n\}$) is equivalent to $\{$f(x_1),f(x_2),...,f(x_n)$\}$).

Figure 6 shows the HorseIR program for the example query in Figure 4a with a scalar UDF. The HorseIR code consists of a module with two methods: the SQL component is translated to the main method, and the UDF is translated to the method `calcRevenueChangeScalar` which takes two arrays of type float as input and returns the resulting product. This method is called by the main method, which otherwise is the same as we have already seen in Figure 2.

3.4 HorsePower Optimizations

HorsePower performs actual compiler-based optimizations when translating a HorseIR program to target C code. Optimizing HorseIR programs is challenging since HorseIR provides a large set of array-based built-in functions and a rich set of data types inherited from database systems. In the following, we will first look at optimizations within one method, and then see how we facilitate cross-optimization through method inlining.

3.4.1 Optimizations within Methods

The optimization techniques used in HorseIR within a method are loop-fusion and pattern-based fusion, and have been introduced in [8], and have been extended in the current HorsePower system. Both are optimization techniques that span multiple statements and built-in functions within a single method.

Automatic loop fusion. This technique is used to fuse array-based built-in functions and generate efficient C code. This approach can be employed across functions that perform element-wise operations on arrays, such as arithmetic operations, boolean selections, reduction, as well as several database-related operations. For example, the HorseIR function `@like`, which is akin to the SQL `LIKE` operator used for text pattern matching, returns a boolean vector that can be fused with subsequent HorseIR built-in functions. Loop-fusion is often used in complex `WHERE` clauses that contain several predicate conditions, but also beyond. For that, HorseIR first builds a data dependence graph across all the statements within a method. Statements which can be fused are then identified by a well-defined dataflow analysis. These statements are later compiled to efficient parallel C code. In fact, the example code that we saw in Figure 3 was generated after generating the data dependence graph that allowed to determine which operations can be fused. A detailed description of the principles can be found in [8]

Pattern-based fusion. A *code pattern* is a pattern that a compiler can recognize within a code snippet, then rewrite it into efficient code following some templates. Therefore, by adding patterns and associated templates to translate a pattern in an optimal way, a compiler's optimization repertoire can be extended, especially to benefit from any domain-specific application characteristics. While HorsePower has extended the pattern repertoire of [7], in the interest of brevity and the fact that their implementation details are more relevant to the core compiler research community, we have omitted the pertaining discussion from this paper.

3.4.2 Cross Optimizations By HorsePower

Considering SQL statements with embedded UDFs, as we have seen in Section 3.3, both the SQL and the UDF part are independently translated into HorseIR and the resulting code then merged to create a main method which calls the method representing the UDF.

For our running example, Figure 6 shows the merged code with the clear separation of the SQL-based and the UDF-based parts. If we were to optimize both parts independently using loop fusion and pattern-based fusion as just discussed, the overall result would be sub-optimal. In fact, if we look at the dependence graph for this program on the left side of Figure 7 (with S_0 to S_4 depicting the statements in the code), we can see that the optimization opportunities are now separated into three snippets: before, after, and in the method being called in the statement S_3. The snippets have to be optimized individually because the content of the statement S_3 is invisible to the rest of the code.

Thus, statements S_1 and S_2 of the main method need to be evaluated and intermediate results `t4` and `t5` cannot be eliminated as the method `calcRevenueChangeScalar` requires their actual values to be passed as parameters. Furthermore, the return value of the method needs to be materialized to be assigned to `t6` which is then the input of statement S_4. This means the potential scope for fusion is significantly reduced leading to more intermediate results.

Therefore, we do not optimize the individual parts independently, but aim at a holistic optimization. The idea to enable this cross-optimization is conceptually simple: *inlining*. This involves replacing the method calls within the main method by the corresponding code segments that constitute the method that is being called. This modified version of the HorseIR program will now have fewer method calls and is conducive to the fusion optimizations that we discussed in Section 3.4.1.

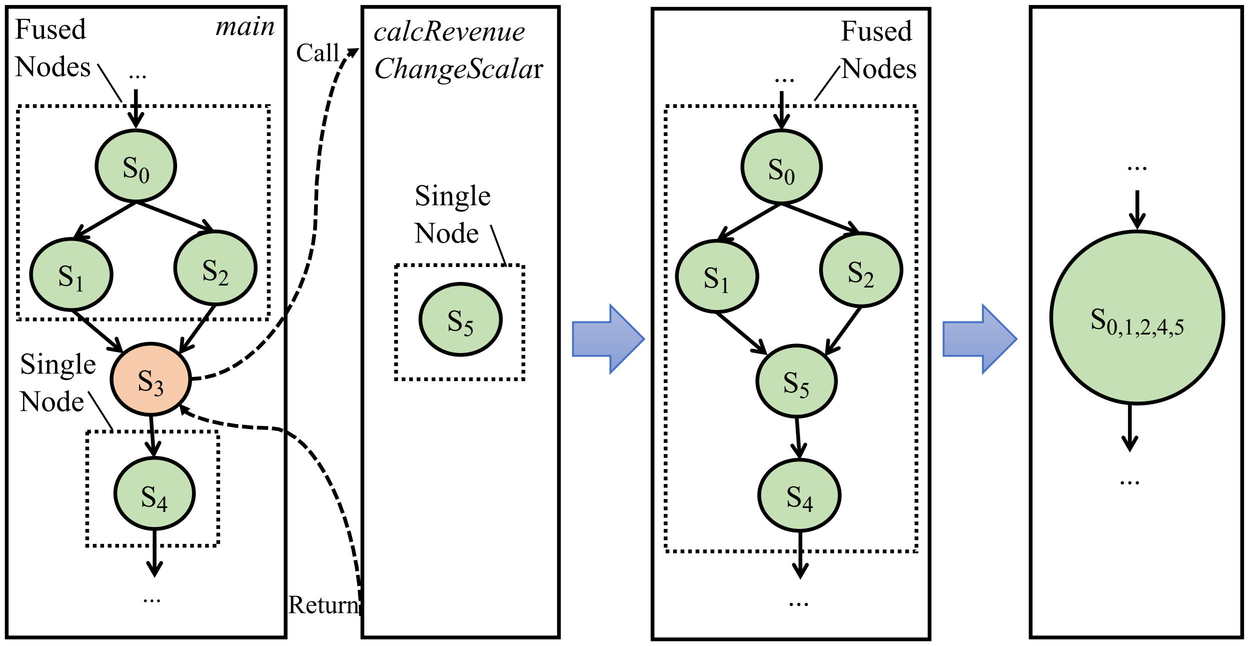

Figure 7: Dependence graphs for the example in Figure 6 to show that method inlining helps explore more opportunities for automatic loop fusion

For our example program in Figure 6 this means the code of `calcRevenueChangeScalar` can be inlined into the main method with the generated HorseIR being almost the same as the one in Figure 2 except for possibly different variable names. As a result, a dependence graph can be built across the main method, as illustrated on the right side of Figure 7, allowing for loop fusion across all statements and generating a single loop of all tasks. In particular, the boolean predicate statement (S_0), the compress statements (S_1 and S_2), the multiplication statement (S_5), and the reduction statement (S_4) can be fused together when generating the C code in Figure 3. The core computational logic of the program is now consolidated into a single loop which performs the tasks of selecting relevant price and discount values as well as calculating the net revenue change. This code is efficient compared to the original version without method inlining, as it avoids the materialization of any intermediate results introduced by UDF invocations.

In some scenarios method inlining offers additional optimization opportunities, such as the elimination of unused computations. For example, consider a scenario where a table UDF computes and returns two columns as part of its invocation, but the enclosing SQL query itself uses only one of those two columns. HorsePower will employ the *backward slicing* technique [34] to avoid the computation of the unused column in the table UDF.

While performing inlining, to respect the pass-by-value convention for parameter passing, a copy of the object used as the parameter will be generated if the parameter is found to be modified inside the original callee method. This ensures that inlining does not result in any unintended data modifications to the objects inside the method that was making the call. Further, if inlining results in any variable name conflicts, they are resolved by assigning new but unique variable names. Finally, an inlined method is removed if it can be inlined in all the code locations where it is called.

4 Evaluation

In this section we present the evaluation result of our framework for SQL queries, MATLAB programs, and SQL queries with analytical UDFs written in MATLAB.

4.1 Overview

Setup for experiments. The experiments are conducted in a multi-socket multi-core server equipped with 4 Intel Xeon E7-4850 2.00GHz (total 40 cores with 80 threads, and 24 MB of shared L3 CPU cache) and 128 GB RAM running Ubuntu 18.04.4 LTS. We use GCC v8.1.0 to compile HorseIR source code with optimization options `-O3` and `-march=native`; MonetDB version v11.35.9 (Nov2019-SP1) and NumPy v1.13.3 along with Python v2.7.17 interpreter for embedded Python support in MonetDB; and MATLAB version R2019a. The response time is measured only for the core computation, and excludes the overhead for parsing SQL, plan generation, compilation, and serialization for sending the results to the client. Scripts and data used in our experiments can be found in our GitHub repository[4].

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 stabilize showing that all data has been brought from disk to main memory by then. Additionally, for HorsePower we measure the compilation time for compiling the generated C code to executable and report the average of 10 runs.

Benchmarks. The performance of the HorseIR execution environment and the advantage of loop-fusion and pattern-based compiler-based optimization techniques for some of the queries of the TPC-H benchmark have already been analyzed in detail in previous work [7]. While HorsePower provides now support for a much larger set of SQL queries and has refined its optimization strategies, the principle findings were already presented in [7]. Therefore, our evaluation focuses on evaluating MATLAB programs and SQL queries with UDFs.

MATLAB. In order to understand the performance implications of using HorsePower for executing non-SQL based data analytics, we use the `Black-Scholes` algorithm from the PARSEC benchmark suite v3.0 [5], and the `Morgan` algorithm [9] from a finance application.

- Black-Scholes is used in finance to compute the price variation of European options over time. There are two UDFs, *BlackScholes* and *CNDF* (standardized Cumulative Normal Distribution

[4]`https://github.com/Sable/edbt21-analysis`

Function), where BlackScholes is the main function for computing option prices. This algorithm is fully vectorizable, and can be efficiently written using array programming. As the original implementation[5] is in C, we reimplemented this algorithm as a MATLAB function.

- Morgan contains a main function *morgan* and a function *msum*. It contains a wide range of element-wise computations and declares several local variables. Instead of using matrices as input in its original implementation, we adapt it to using vectors which can represent two columns from one table. We implement this algorithm as MATLAB functions and have set the first parameter to 1000 (i.e. N=1000).

SQL with UDF benchmarks. In order to extensively test HorsePower's performance on database queries integrating analytical UDFs, we created the following benchmark.

In the first benchmark, we create SQL queries with UDFs based on the TPC-H benchmark as proposed by Froid [30]. TPC-H [35] contains a suite of 22 SQL queries, from simple to quite complex covering a wide spectrum of SQL constructs with different performance implications, such as the types of condition predicates and joins, the sizes of tables, the number of columns, and the number of records that are returned. However, TPC-H is a pure SQL benchmark and does not include any UDFs. Froid [30] has rewritten these queries so that some of the semantics, such as checking certain conditions, are now within UDFs that are then embedded in the modified SQL statement. In most of these queries, these UDFs are part of the SELECT or the WHERE clause.

In our second benchmark we embed the Black-Scholes algorithm in form of UDFs into SQL queries. We created both scalar and table UDF variations as well as designed several enclosing SQL statements that offer the different potential for optimizations.

4.2 MATLAB Benchmarks

In this section we analyze how well HorsePower performs in executing code originally written in MATLAB by taking the Black-Scholes and Morgan algorithms as described in Section 4.1. Recall that HorsePower first translates MATLAB code into TameIR using the McLab framework, then translates the TameIR program into HorseIR, and finally builds low-level C code. In our experiments, we compare the following:

- We execute the original MATLAB program using the MATLAB interpreter with default settings.
- We compile the HorseIR program generated from the MATLAB code into C code without any of the optimizations that we mentioned in Section 3.4 such as loop fusion. We refer to this version as HorsePower-Naive. As such, it is likely to produce a similar amount of intermediate results as the MATLAB interpreter.
- We compile the HorseIR program into C code with all optimizations enabled. We refer to this version as HorsePower-Opt.

Table 1 shows the execution times for MATLAB and for the two HorsePower versions with different sizes of the Black-Scholes and Morgan tables, respectively (each from 1 to 8 million rows). We also

[5]Downloadable PARSEC package: `http://parsec.cs.princeton.edu/`

Table 1: Speedup of HorsePower over MATLAB in execution time using Black-Scholes and Morgan (in milliseconds)

Black-Scholes

Size	MATLAB	HorsePower			
		Naive	Speedup	Opt.	Speedup
1M	61	66	0.92x	7	9.34x
2M	145	137	1.06x	14	10.17x
4M	491	463	1.06x	49	10.12x
8M	1009	1384	0.73x	117	8.60x

Morgan

Size	MATLAB	HorsePower			
		Naive	Speedup	Opt.	Speedup
1M	83	80	1.04x	25	3.28x
2M	221	178	1.24x	49	4.48x
4M	563	356	1.58x	110	5.11x
8M	1423	667	2.13x	202	7.05x

indicate the speedup of HorsePower over MATLAB in execution time for both HorsePower versions. Note that the MATLAB interpreter does not allow to control the number of threads and instead aims in using all physical threads. For HorsePower the results shown are with 40 threads.

In Black-Scholes, we observe that the execution times for MATLAB and HorsePower-Naive are similar, with slightly better performance for MATLAB. We believe this is due to MATLAB having more efficient library functions that work well even in an interpreter mode. As can be seen in Morgan, the execution times for HorsePower-Naive are faster than MATLAB. We believe the reason is our efficient parallel implementation of built-in functions, such as the cumulative sum (*cumsum*). When comparing with HorsePower-Opt, MATLAB is significantly slower in both benchmarks. The reason is that HorsePower-Opt optimizations, in particular loop fusion, are able to avoid many intermediate results, speeding up the computation by a large margin. For both comparisons, the size of the data set plays a minor role.

In summary, we can see that HorsePower is a promising approach to execute data analytics tasks in an efficient manner. This is due to its data-centric IR that makes it possible to exploit data-centric compiler optimization techniques.

4.3 SQL and UDF Benchmarks: TPC-H

This is the first of two sections to compare the performance of HorsePower and MonetDB in executing SQL statements with embedded UDFs. As mentioned in Section 4.1, for HorsePower the UDF is written in MATLAB, for MonetDB in Python using the NumPy library, with an effort to have similar code within the UDF.

Froid [30] proposed a whole range of queries derived from the TPC-H benchmark in which part of the SELECT or WHERE clauses are outsourced into a UDF. In all cases, these are scalar UDFs. For instance, modified q6 is very similar to our example query of Figure 2a, it just contains many more conditions involving more attributes. Some of these UDFs have embedded SQL statements. However, the McLab framework that we use to translate MATLAB programs currently only supports pure MATLAB programs. Thus, we excluded those unsupported queries and present results only for queries q1, q6, q12, q14, and q19.

Table 2: Speedup (SP) of HorsePower over MonetDB in execution time using the modified TPC-H benchmarks with UDFs

Thread	MonetDB (ms)					HorsePower (ms)									
	q1	q6	q12	q14	q19	q1	SP	q6	SP	q12	SP	q14	SP	q19	SP
T1	16853	48832	137195	1040	69045	3799	4.44x	392	125x	900	152x	904	1.15x	858	80.5x
T2	11439	48930	140118	989	76153	2548	4.49x	220	223x	493	284x	558	1.77x	512	149x
T4	7304	48247	144962	846	75012	2897	2.52x	130	372x	340	426x	446	1.90x	346	217x
T8	5724	47775	143714	773	72124	3316	1.73x	56	853x	300	479x	396	1.95x	364	198x
T16	3549	46996	142819	764	69997	2620	1.35x	42	1124x	238	600x	318	2.40x	245	286x
T32	2502	44636	140438	750	64267	1883	1.33x	45	1000x	170	826x	216	3.48x	209	307x
T64	2227	N/A	138526	743	65603	2256	0.99x	26	N/A	141	984x	197	3.77x	199	329x
HorsePower Compilation Time (ms):						358		250		326		542		354	

Table 2 shows the execution times of these queries with a different number of threads using HorsePower and MonetDB. For HorsePower the results show the execution times after all optimizations have been performed (merge with consecutive loop and pattern fusion).

When first looking only at MonetDB we can see that execution times are relatively low for some queries and improve with an increasing number of threads considerably (q1 and q14), but are high for others with little benefit of parallelization (q6, q12, q19). The reason is that in these queries, the UDF is in the WHERE clause and MonetDB has to perform costly data conversion when sending the entire database columns as arrays to the Python interpreter in order to execute the UDF. MonetDB is able to use zero-copy transfer for data types where the database system uses the same main-memory representation as Python. But for strings, it needs to convert the data to a different format as the database internal and the Python formats are incompatible. This data conversion seems to not be parallelized to multiple threads, making it the predominant factor of the execution. In q1 and q14, the UDFs are in the SELECT clause (where data sizes are smaller as they got reduced due to the selection that was already executed), and do not require any string conversions.

HorsePower has overall much better performance for all queries, being under 1 second for all queries except q1, and can always improve execution times by increasing the number of threads. As no data conversion is necessary it is orders of magnitude faster than MonetDB for queries q6, q12, and q19. We can observe here the advantage of having a unified execution environment that has translated both the UDF part and the SQL part to a single HorseIR program with its own data structures. But we also observe significant improvements for q1 and q14. These are due to the unified optimization across the HorseIR code generated from SQL and UDF.

4.4 SQL and UDF Benchmarks: MATLAB

With the purpose of studying the performance of queries with UDFs derived from MATLAB benchmarks executed in HorsePower and MonetDB, we set up the benchmarks as follows:

- In the HorsePower version, the MATLAB program and the SQL statement were both independently translated to HorseIR, the code then merged and optimized.
- In the MonetDB version, we implemented Black-Scholes as Python UDFs that compute the `optionPrice`, and wrote SQL queries to invoke these UDFs.

- For both versions, we created a whole set of enclosing SQL statements for being able to test variations of the UDFs.

Table 3: Black-Scholes execution time Python vs. HorseIR

Python (T1)	HorsePower			
	Naive(T1)	Speedup	Opt(T1)	Speedup
514.78	577.4	0.89x	247.9	2.08x

The Python UDF is implemented with the NumPy library using the same array programming style as the MATLAB UDF. In fact, for the Black-Scholes benchmark, each array operation in MATLAB has an equivalent array operation in NumPy. In order to understand the implication of having a Python UDF for MonetDB and a HorseIR program for HorsePower, Table 3 shows the execution time of the Black-Scholes benchmark for the dataset in this section both using a Python program and using HorseIR (both naive and optimized). Execution is in one thread because NumPy does not support multi-threading for operations in the benchmark. Similar to what we have seen with our analysis with MATLAB, a naive usage of HorseIR provides similar execution time as Python; performing optimizations achieves a speedup of 2.

Table 4: Performance comparison between HorsePower (HP) and MonetDB (MDB) for variations in Black-Scholes, including the speedup (SP) and the compilation time (COMP) for compiling the generated C code to the executable in HorsePower

UDF	Selec.	Table UDF (ms)							Scalar UDF (ms)						
		T1			T64			HP	T1			T64			HP
		MDB	HP	SP	MDB	HP	SP	COMP	MDB	HP	SP	MDB	HP	SP	COMP
bs0_base	100.0%	927.5	249.8	3.71x	774.0	7.09	109x	351	670.0	249.5	2.69x	696.5	7.06	98.6x	375
bs1_high	0.2%	926.4	256.2	3.62x	818.0	7.62	107x	372	6.10	0.32	19.1x	6.55	0.13	50.4x	380
bs1_med	50.9%	914.7	262.4	3.49x	794.2	12.2	65.0x	373	308.6	86.6	3.57x	272.2	2.51	108x	382
bs1_low	99.8%	929.7	266.4	3.49x	832.9	14.6	57.0x	375	725.4	169.6	4.28x	645.4	4.90	132x	380
bs2_high	0.2%	895.6	4.67	192x	791.5	0.70	1131x	252	4.29	4.59	0.93x	3.52	0.63	5.59x	256
bs2_med	50.9%	912.5	8.24	111x	811.6	4.22	192x	252	13.4	8.20	1.63x	4.16	4.29	0.97x	252
bs2_low	99.8%	916.4	11.0	83.7x	820.4	6.64	124x	252	15.9	10.95	1.45x	5.11	5.95	0.86x	251
bs3_high	10.0%	911.8	259.0	3.52x	824.4	10.1	81.6x	379	673.8	179.3	3.76x	623.2	7.69	81.0x	374
bs3_med	49.5%	906.5	263.7	3.44x	831.6	13.3	62.7x	377	678.9	184.1	3.69x	631.6	11.3	56.1x	375
bs3_low	90.0%	879.1	262.5	3.35x	793.6	13.7	57.8x	377	685.4	182.6	3.75x	641.7	12.8	50.1x	377

In order to test the two different types of programming approaches that databases support, we created two variants of the SQL function, implemented as UDFs. In one variant, we created a *scalar UDF* that returns just the computed `optionPrice` to the calling SQL.

```
CREATE SCALAR UDF bScholesUDF(spotPrice, ..., optionType)
{
  import blackScholesAlgorithm as bsa
  return 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)
  return [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 that had different columns in the `SELECT` and `WHERE` clauses. Further, for each of the variation, we modified the values associated with the conditional predicates in the selection (`WHERE` clause), so that the selectivity varies between high, low, and medium. In a highly selective condition, only a 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. We describe these queries in more detail in the next subsections.

Table 4 shows the selectivity for variations derived from Black-Scholes, covering low, medium, and high selectivity, and the execution time of MonetDB and HorsePower for the table and scalar UDFs respectively. Due to the constraint of space limit, we only present the result for 1 thread (T1) and 64 threads (T64). The COMP column indicates the compilation time spent in HorsePower. We can see that it is between 250 and 380 ms and little impacted by selectivity. In the next subsections, we discuss the results in detail.

Base query. 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.

```
-- 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));
```

We can first observe that for MonetDB multi-threading has little impact on its performance. In contrast, HorsePower benefits a lot. Thus, when looking at one thread, HorsePower is around 3x to 4x speedup compared to MonetDB, while it is around 100x speedup with 64 threads. The main reason is that a large part of the execution is in the Black-Scholes algorithm due to the large data input in this query, and Python is not multi-threaded, i.e., this part of the execution in MonetDB always runs within one thread. In contrast, HorsePower can create optimized parallel code.

However, HorsePower has even significant benefits with a single thread. In fact, with one thread, the execution time with 249.8 ms is basically equivalent to executing the Black-Scholes algorithm alone, without the SQL part, which is 247.9ms as shown in Table 3. For MonetDB, executing the algorithm within an SQL statement in the form of a Python UDF is with 927.5 ms nearly double as long as executing the Python function in standalone mode with 577.4 ms. As the SQL part of this query is straightforward, the reason for this performance penalty in MonetDB must be the communication between its SQL engine and the Python UDF interpreter.

Variation 1. 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.

```sql
-- Query, bs1_high, Scalar UDF
SELECT spotPrice, optionType,
  bScholesUDF(spotPrice,...,optionType)
  AS optionPrice
FROM blackScholesData
WHERE spotPrice < 50 OR spotPrice > 100;

-- Query, bs1_high, Table UDF
SELECT spotPrice, optionType, optionPrice
FROM bScholesTblUDF
  ((SELECT * FROM blackScholesData))
WHERE spotPrice < 50 OR spotPrice > 100;
```

Looking at the performance numbers, we can see that for one thread, HorsePower's speedup over MonetDB is at least 3.5x for both scalar and table UDFs, and for 64 threads at least 50x.

For the SQL using scalar UDF, MonetDB can infer that the conditions are placed on the input column and then discard the records that do not qualify before processing the UDF. This approach follows the traditional database optimization technique of applying high selectivity operations first. As HorsePower relies on MonetDB for database execution plans, it is similarly impacted by the plans generated by MonetDB for table UDF based queries. This results in HorsePower's own table UDF based queries costing more than its scalar versions. However, unlike MonetDB, HorsePower benefits from being able to avoid data copies and conversions as well as from generating parallelized code for UDFs, thus expanding this performance gap when the number of threads increases.

Variation 2. 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. As can be seen in the performance numbers, HorsePower achieves only a speedup of at most 2x with one thread and at most 5.5x with 64 threads for the scalar UDFs, but has scale-up of at least 83x with table UDFs, going up to over 1000x for one UDF with 64 threads.

```sql
-- Query, bs2_high, Scalar UDF
SELECT spotPrice, optionType
FROM (
  SELECT spotPrice, optionType, bScholesUDF(spotPrice, ..., optionType) as optionPrice
  FROM blackScholesData
) AS tableBS
WHERE spotPrice < 50 OR spotPrice > 100;

-- Query, bs2_high, Table UDF
SELECT spotPrice, optionType
FROM bScholesTblUDF
  ((SELECT * FROM blackScholesData))
WHERE spotPrice < 50 OR spotPrice > 100;
```

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, HorsePower, being an integrated system, can avoid the computation of `optionPrice` by using a backward slice. However, with a table UDF, MonetDB is unable to avoid this computation as there is no way for it to pass this optimization information to the UDF interpreter. On the other hand, HorsePower uses method inlining and backward slicing to remove this computation, offering a huge advantage.

Variation 3. The last variation, `bs3_*` applies a predicate condition on `optionPrice`. As this is a column computed by the UDFs, both the systems have to process the UDFs across all input records before discarding records that do not qualify, providing limited opportunities for optimization.

```
-- Query, bs3_high, Scalar UDF
SELECT spotPrice, optionType
FROM (
  SELECT spotPrice, optionType, bScholesUDF(spotPrice, ..., optionType) as optionPrice
  FROM blackScholesData
) AS tableBS
WHERE optionPrice > 15;

-- Query, bs3_high, Table UDF
SELECT spotPrice, optionType
FROM bScholesTblUDF
  ((SELECT * FROM blackScholesData))
WHERE optionPrice > 15;
```

Looking at the performance numbers, we can see that HorsePower has speedups of around 3.5x for both scalar and table UDFs with one thread and between around 50x and 80x for 64 threads. In this scenario, both execute the full UDFs before applying the condition. HorsePower has better performance than MonetDB simply because HorsePower can save the data movement between the UDF and the query while it is mandatory for MonetDB to have data conversion between the database and the UDF engine (Python). With more threads, the performance becomes worse since the data movement is sequential and takes most of the time in the whole execution pipeline.

In summary, we observed that while modern RDBMS implementations provide a convenient way to integrate UDF usage into database queries, their resulting execution plans are often sub-optimal due to their black-box integration with the UDF language's execution environment. On the other hand, our advanced analytical system HorsePower optimizes both SQL and statistical language implementations using a common IR based environment. This capability also allows HorsePower to optimize complex analytical tasks that include both SQL and UDF in query in a holistic manner, providing better performance than popular RDBMS approaches.

5 Related Work

Intermediate representations and compiler techniques have been applied by others to improve the performance of database queries. Voodoo [27] is a declarative intermediate algebra designed with a set of vector operators. An SQL query is first compiled to Voodoo code in a database, for example, MonetDB [16]. Then, the Voodoo code is compiled to efficient parallel GPU code. HyPer [25] delegates code generation to LLVM, a low-level intermediate representation. It employs a push-based query engine approach [31] in order to generate efficient LLVM code with less data materialization. This approach also applies to LegoBase [18, 32], whereby LegoBase offers multiple IRs for implementing an efficient query compiler.

However, there is little research in these systems extending to support UDFs within the database queries.

When integrating data analytics into a database system, a direct approach is to develop SQL-like languages to support data analytics by adding new features, such as SciQL [37] and SciDB [6]. Similar efforts have been made to introduce new facilities into database systems to support data analytics, such as for machine learning computations [12, 15, 23, 17]. These solutions are somewhat limited as new features or facilities have to be tailored for specific domains. In contrast, HorseIR supports the widely adopted languages SQL and MATLAB.

Froid [30] shows a holistic optimization solution by transforming simple UDF to relational code. Thus, the existing query optimizer can be utilized for optimization of the execution plan. However, this approach is limited as not all UDFs are translatable to a relational operator.

Weld [26] presents its IR (WeldIR) to support the code generation from various source languages. WeldIR is able to handle database queries and call UDFs written in C code. However, in contrast to HorsePower that automatically optimizes across different source languages, such capabilities have not been implemented by Weld.

Lara [20] is a domain-specific language tailored for relational algebra and UDFs. Its code is first compiled to an IR which is able to inspect UDFs by collecting necessary information from UDFs. Thus, Lara can optimize such transparent UDFs together with its IR code. This is different from our HorsePower which compiles database queries and UDFs to its common IR with holistic optimizations enabled.

Finally, the most popular approach has been to integrate an execution environment for popular analytical tools and languages inside DBS. This "black-box" approach is what MonetDB provides with an embedded Python interpreter and UDF constructs [29, 28]. However, as we saw in the evaluations, the data movement from database to Python is expensive due to the data copy and conversion between two different systems. Further, as also demonstrated in our evaluations, such a black-box implementation results in sub-optimal execution plans, reducing the optimization opportunities across the DBS engine and the UDF execution environment. Being a unified system that is capable of translating both SQL and the analytical languages used for UDFs into a common IR, HorseIR can overcome these hurdles, providing a holistic optimization and execution environment.

6 Conclusions

In this paper we presented `HorsePower`, a system with the capabilities of improving the performance of database queries, MATLAB programs, and database queries with analytical UDFs. We explored MATLAB as a prototype language for supporting database UDFs, which can work with database queries seamlessly via HorseIR. At the core of the system, we introduced a new translator for generating HorseIR code from MATLAB. We then developed an approach that combines HorseIR code from both MATLAB and SQL, and performs sophisticated fusion-based optimizations. Based on the result of thorough experiments, we demonstrated that our system is able to generate efficient code for SQL queries, MATLAB programs, and both mixed. With the promising evaluation results, in the future, we would like to (1) extend McLab, the MATLAB compiler framework, to support more MATLAB features, (2) explore more optimizations such as efficient multiple joins, and (3) experiment on parallel accelerators, such as GPUs.

References

[1] MATLAB. `https://www.mathworks.com`. [Last accessed in October 2020].

[2] McLab: A Framework for Dynamic Scientific Languages. `http://www.sable.mcgill.ca/mclab/`. [Last accessed in October 2020].

[3] R Project. `https://www.r-project.org`. [Last accessed in October 2020].

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

[5] 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, 2008.

[6] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD*, pages 963–968, 2010.

[7] H. Chen, J. V. D'silva, H. Chen, B. Kemme, and L. Hendren. HorseIR: Bringing Array Programming Languages Together with Database Query Processing. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, (DLS'18)*, pages 37–49, 2018.

[8] H. Chen, A. Krolik, B. Kemme, C. Verbrugge, and L. Hendren. Improving Database Query Performance with Automatic Fusion. In *Proceedings of the 29th International Conference on Compiler Construction, (CC'20)*, pages 63–73, 2020.

[9] 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.

[10] J. V. D'silva, F. De Moor, and B. Kemme. AIDA - Abstraction for Advanced In-Database Analytics. *PVLDB*, 11(11):1400–1413, 2018.

[11] A. W. Dubrau and L. J. Hendren. Taming MATLAB. In *OOPSLA*, pages 503–522, 2012.

[12] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD*, pages 325–336, 2012.

[13] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[14] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance Tradeoffs in Read-optimized Databases. In *PVLDB*, pages 487–498, 2006.

[15] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[16] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[17] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao. Declarative recursive computation on an RDBMS. *Proc. VLDB Endow.*, 12(7):822–835, 2019.

[18] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.

[19] V. Kumar and L. J. Hendren. MIX10: compiling MATLAB to X10 for high performance. In *OOPSLA*, pages 617–636, 2014.

[20] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB*, 12(11):1553–1567, 2019.

[21] J. Lajus and H. Mühleisen. Efficient Data Management and Statistics with Zero-Copy Integration. In *International Conference on Scientific and Statistical Database Management*, pages 12:1–12:10. ACM, 2014.

[22] X. Li and L. J. Hendren. Mc2FOR: A Tool for Automatically Translating MATLAB to FORTRAN 95. In *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pages 234–243, 2014.

[23] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. Scalable Linear Algebra on a Relational Database System. In *33rd IEEE International Conference on Data Engineering, (ICDE'17)*, pages 523–534, 2017.

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

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

[26] S. Palkar, J. J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. P. Amarasinghe, S. Madden, and M. Zaharia. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB*, 11(9):1002–1015, 2018.

[27] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 9(14):1707–1718, 2016.

[28] M. Raasveldt. Integrating Analytics with Relational Databases. In *Proceedings of PhD Workshop co-located with VLDB'18*, 2018.

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

[30] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. A. Galindo-Legaria, and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB*, 11(4):432–444, 2017.

[31] A. Shaikhha, M. Dashti, and C. Koch. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28:e10, 2018.

[32] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to Architect a Query Compiler, Revisited. In *SIGMOD*, pages 307–322, 2018.

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

[34] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3), 1995.

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

[36] B. Woody, D. Dea, D. GuhaThakurta, G. Bansal, M. Conners, and T. Wee-Hyong. *Data Science with Microsoft SQL Server 2016*. Microsoft Press, 2016.

[37] Y. Zhang, M. L. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *SIGMOD*, pages 1049–1052, 2013.