HorsePower: Accelerating Database Queries for Advanced Data Analytics

Hanfeng Chen, Joseph V. D'silva, Laurie Hendren, Bettina Kemme
McGill University
hanfeng.chen@mail.mcgill.ca,joseph.dsilva@cs.mcgill.ca,hendren@cs.mcgill.ca,kemme@cs.mcgill.ca

ABSTRACT

The rising popularity of data science has resulted in a challenging interplay between traditional declarative queries and numerical computations on the data. In this paper, we present and evaluate the advanced analytical system HorsePower that is able to combine and optimize both programming styles in a holistic manner. It can execute traditional SQL-based database queries, programs written in the statistical language MATLAB, as well as a mix of both by supporting user-defined functions within database queries. HorsePower exploits HorseIR, an array-based intermediate representation (IR), to which source programs are translated, allowing to combine query optimization and compiler optimization techniques at an intermediate level of abstraction.

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 based on languages such as Python, or the statistical languages MATLAB [1], and R [3]. However, as the size of the data increases, the expensive data movement between DBS and analytics tools can become a severe bottleneck.

Integrating analytical capabilities into the DBS avoids such expensive data exchange. A common approach is to use user-defined functions (UDFs) that are embedded in SQL queries [13]. For example, MonetDB supports UDFs written in Python, that are executed by a Python language interpreter that is embedded inside the DBS engine.

While no data transfer is needed with this approach, there are still two separate execution environments, one being the SQL execution engine, the other the programming language execution environment. This can lead to costly data format conversion. Furthermore, the SQL and the UDF components of the query are each individually optimized by their respective execution environments, without the consideration of any holistic optimization across the entire task.

To address these issues, we propose `HorsePower`, an advanced analytical SQL system, which provides a holistic solution to integrate UDFs in SQL queries. The system is based on HorseIR [5], an array-based intermediate representation (IR) language which was developed to explore the usage of compiler optimizations for query execution. Chen et al. [5] 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. Using arrays to represent database columns, HorseIR follows conceptually the data model of column-based DBS, which has been proven to be effective for data analytics tasks.

HorsePower extends the idea to a full-fledged execution environment for data analytics. Additionally to supporting plain SQL queries, HorsePower also supports functions written in MATLAB, a popular high-level array language widely used in the field of statistics and engineering. HorsePower can take stand-alone functions written in MATLAB and translate them to HorseIR, of have these functions be embedded in SQL queries and then translate all into a single HorseIR program, before optimizing and compiling the code 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 optimizations across both the declarative and functional parts of the query.

The contributions of this paper are thus as follows:

- We present HorsePower, an advanced analytical system, that extends the approach proposed in [5] 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.

2 BACKGROUND

2.1 HorseIR: an Array-based IR for SQL

Recent years have seen the development of modern query compilers that translate an SQL query into an intermediate representation (IR) before target code is generated from the IR, making it possible to leverage any existing code optimizations available within the IR platform.

In this context, HorseIR [5] was developed as a high-level IR specifically for database applications [7]. 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. [5] took the execution plans generated by the column-based database system HyPer [11], 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, allowing for an intuitive merge of chained operations and

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

```
1  module ExampleQuery{
2   def main(): table{
3    ...
4    // assume t1, t2 are references to l_price/l_discount columns
5    t3:bool = @geq(t2, 0.05);
6    t4:f64  = @compress(t3, t1);
7    t5:f64  = @compress(t3, t2);
8    t6:f64  = @mul(t4, t5);
9    t7:f64  = @sum(t6);
10   ...}}
```

Figure 1: Example query and its HorseIR program

thus, avoiding intermediate results. Thus, optimizations developed for array-based programming languages can be exploited to improve query performance.

Example The top of Figure 1 shows a simplified version of Query 6 of the TPC-H benchmark [16] computing the change in total revenue given prices and discounts from the table `lineitem`. A basic translation into a HorseIR program prior to performing any optimizations is shown at the bottom of Figure 1, outlining only the part of the code that performs the actual relational operators. We assume that arrays `t1` and `t2` represent the price and discount columns. The program computes the WHERE condition (`@geq`), which returns a boolean vector of the same length as `t2` with `true` values in all rows that fulfill the condition. The function `@compress` then extracts from both `t1` and `t2` the rows for which the boolean vector has a `true` value. The output are "compressed" vectors with relevant rows, over which then the aggregation is performed in two steps.

HorseIR Optimizations As can be seen, such an approach can generate a fair amount of intermediate results (arrays `t3` to `t6` in the example). If lines 5 to 9 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. To do such fusion, HorseIR first builds a data dependence graph across all the statements. Statements which can be fused or follow a pattern, are then identified by a well-defined data flow analysis, and compiled together to efficient C code. For our example, the resulting sequential C code would look similar to

```
...
revenue = 0;
for(i = 0; i < numRows; i++)
{ if(t2[i] >= 0.05) revenue += t2[i] * t3[i]; }
...
```

Although the example C code does not convey it explicitly, behind the scenes, HorseIR uses OpenMP to compile the program into a parallel implementation, as outlined in [5].

2.2 Traditional Database UDFs

A UDF is a high-level language function embedded within an SQL statement, and is used to offload partial computation into are more concise language than SQL, or provide additional functionality. To support UDFs, the database system integrates the language runtime environment into the DBS (such as the Python interpreter in MonetDB [13]). We will focus only on *Scalar UDFs* and *Table UDFs*, as

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

```
SELECT SUM(RevChangeSclr(l_price,l_discount)) AS RevChange
FROM lineitem WHERE l_discount >= 0.05;
```

Figure 2: Rewriting the example query with a scalar UDF

these are the most commonly employed types of UDFs and also the ones supported presently in HorsePower.

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 2 shows a scalar UDF which performs the multiplication that was originally part of the `SELECT` clause in Figure 1. In a column-based database system, the execution of such a query first evaluates the `WHERE` clause on `l_discount`, returning a boolean vector. Then, the database applies the corresponding boolean selection on columns `l_discount` and `l_extendedprice`, returning compressed vectors containing the rows where the boolean vector was true. These columns are then given to the UDF as arrays, and the UDF performs an element-wise multiplication on them and produces a result array. This is then the input to the `SUM` operator. Thus, the UDF is only called a single time and works on entire arrays.

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. For an example of a table UDF, we refer to a technical report [6].

Introducing UDFs into queries can bring performance issues. If the data types used by the two execution environments are different, this can introduce a conversion overhead when exchanging data. Further, as UDF languages are typically black-boxes to the database engine, cross optimization attempts are minimal, resulting in sub-optimal execution plans.

3 HORSEPOWER

In this section we present HorsePower, a system 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.

3.1 SQL to HorseIR

While prior work used HyPer's execution plans [11] to translate SQL to HorseIR, HorsePower uses MonetDB's execution plans, as MonetDB supports UDFs and the execution plans contain the relevant UDF information. Our implementation first translates the tree-based plans to JSON objects that are then translated to HorseIR[1].

Furthermore, HorsePower supports a wider range of SQL queries than [5], which did not properly support multi-join queries. This includes all queries of the TPC-H benchmark [16].

3.2 MATLAB to HorseIR

MATLAB is a sophisticated dynamic language which provides numerous flexible language features. In order to transform MATLAB code to HorseIR, as an intermediate step,

[1]HorsePower could generate it own execution plans. However, as the traditional query optimization techniques are not the focus of our research, we preferred to integrate the already optimized execution plans generated by existing DBS.

HorsePower calls upon the McLab framework [2] which translates MATLAB programs to its own internal IR, called TameIR, handling MATLAB's many dynamic features and lack of strict typing. Type and shape information for all variables in the program are automatically derived. Furthermore, class program analysis steps, such as constant propagation, are performed to produce optimized TameIR code [9]. TameIR can represent MATLAB's matrix and high-dimension arrays, and currently supports an essential subset of MATLAB array operations.

HorsePower then translates TameIR code to HorseIR. So far, this translator supports a core subset of MATLAB features and built-in functions. It preserves MATLAB pass-by-value semantics but automatically switches to pass-by-reference when it determines that the input parameters are not modified, avoiding data copies. It supports the common control structures `if-else` and `while` with a restriction on the condition which must be a single boolean element. While explicit loop iteration is not supported, MATLAB's array-based built-in functions (which have implicit loop execution) are translated in a straightforward way as similar functions exist in HorseIR. All types supported by TameIR are also supported by HorseIR, however, due to type rule mismatches, input types for some operators are restricted (e.g. because *integer* + *double* returns integer in MATLAB, but double in HorseIR). Finally, the translator requires MATLAB arrays to have the data layout of `1-by-N` instead of `N-by-1`, as the former one 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 execution plans generated by MonetDB, which contain hooks into UDFs with their names, and input and output parameters, but otherwise treat the UDFs as a black-box. HorsePower translates such a plan to HorseIR, where the invocation of the UDF is translated to 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.

HorsePower supports both scalar and table UDFs. In order to make the MATLAB functions conform to the semantic form expected of these types of UDFs, we enforce some restrictions on the MATLAB functions. For instance, we require a function to have one return statement with either a single vector (for scalar UDFs) or a table-like data structure (for table UDFs).

Figure 3 shows the HorseIR program for the example query in Figure 2 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 `RevChangeSclr` 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 1.

3.4 Holistic HorsePower Optimizations

HorsePower performs compiler-based optimizations when translating a HorseIR program to target C code. We have discussed in Sec. 2.1, how *automatic loop-fusion* and *pattern-based*, as introduced in [7] lead to efficient parallel C code.

```
module ExampleQuery{
  def RevChangeSclr(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 = @RevChangeSclr(t4,t5); // S3
     t7:f64 = @sum(t6);               // S4
     ...
}}
```

Figure 3: HorseIR code for the Query in Figure 2

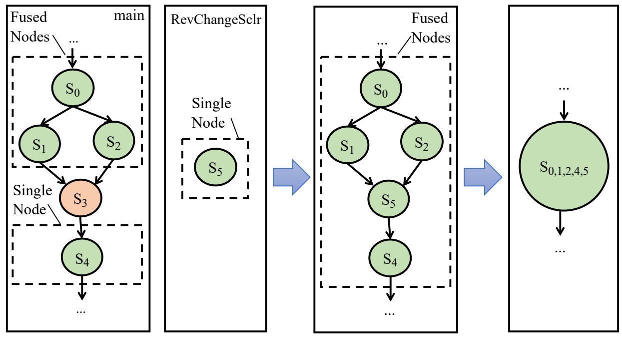

Figure 4: Dependence graphs for the example in Figure 3

However, such optimizations require all statements to be in one method. But when SQL statements have embedded UDFs, the HorseIR code has at least two methods, with a main method calling the method representing the UDF as shown in our example in Figure 3.

If we were to optimize both parts independently using loop fusion and pattern-based fusion, 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 4 (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 `RevChangeSclr` 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 the statement S_4. This means the potential scope for fusion is significantly reduced leading to more intermediate results.

In order to enable a more holistic cross-optimization, we use the concept of *inlining*. This involves replacing the method calls within the main method with the corresponding code segments that constitute the method that is being called. For our example program in Figure 3 this means the code of `RevChangeSclr` can be inlined into the main method with the generated HorseIR being almost the same as the one in Figure 1 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 4, allowing for loop fusion across all statements and generating a single loop of all tasks as outlined in Section 2.1, and avoiding 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 [15] 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 pure MATLAB programs, and for SQL queries with analytical UDFs written in MATLAB. For the latter, we compare it with MonetDB.

The experiments are conducted on a 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. We only consider execution time once data resides in the main memory. We run each test 15 times but only measure the average execution time over the last 10 times. Scripts and data used in our experiments can be found in our GitHub repository[2].

4.1 MATLAB Benchmarks

We first evaluate MATLAB programs in order to understand the performance of using HorsePower for executing non-SQL based data analytics, and use the following benchmarks: the `Black-Scholes` algorithm from the PARSEC benchmark suite v3.0 [4] having two UDFs *BlackScholes* and *CNDF*, and the `Morgan` algorithm [8] from a finance application having a main function *morgan* and another function *msum*. Both contain several element-wise functions and are fully vectorizable.

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. 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 code into C code with all optimizations enabled, referred to as HorsePower-Opt.

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

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

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

Table 1 shows the execution times for MATLAB and for the two HorsePower versions with different sizes of the Black-Scholes tables. We also indicate the speedup of HorsePower over MATLAB in execution time. Note that the MATLAB interpreter uses all physical threads. For HorsePower, we used 40 threads.

The execution times for MATLAB and HorsePower-Naive are similar, with slightly better performance for MATLAB, probably due to MATLAB having more efficient library functions. When comparing with HorsePower-Opt, MATLAB is significantly slower. The reason is that HorsePower-Opt optimizations, in particular loop fusion, are able to avoid many intermediate results. We also observe that the size of the data set plays a minor role.

For Morgan (no table shown due to space limitations) we run experiments up to 8 million rows as well. HorsePower-Naive also provides similar performance to MATLAB with smaller data sizes, but already has a speedup of 2 with 8 million rows. We believe the reason is our efficient parallel implementation of built-in functions, such as the cumulative sum. Again, the optimized version is significantly faster, with a speedup of 7 with 8 million rows.

In summary, HorsePower can execute data analytics tasks in an efficient manner due to its data-centric IR and compiler optimization techniques.

4.2 SQL and UDF Benchmarks: TPC-H

This is the first of two sections to evaluate the performance of HorsePower in executing SQL statements with embedded UDFs, and comparing it with MonetDB.

Froid [14] proposed a whole range of queries derived from the TPC-H benchmark in which part of the SELECT or WHERE clauses, e.g., to check certain conditions, are outsourced into a UDF. In all cases, these are scalar UDFs. For instance, they propose a variation of the q6 of the TPC-H benchmark, which is very similar to our example query of Figure 1, simply containing more conditions.

For MonetDB, we rewrote the queries to use Python-based UDFs, for HorsePower, the UDFs are written in MATLAB. The structure of the programs is very similar for both languages. Some of the proposed UDFs have embedded SQL statements which are currently not supported by the McLab framework that we use. Thus, we excluded those unsupported queries and present results only for queries q1, q6, q12, q14, and q19.

Table 2 shows the execution times of these queries with a different number of threads using HorsePower and MonetDB. 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

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
T8	5724	47775	143714	773	72124	3316	1.73x	56	853x	300	479x	396	1.95x	364	198x
T32	2502	44636	140438	750	64267	1883	1.33x	45	1000x	170	826x	216	3.48x	209	307x

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 observe 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.

4.3 SQL and UDF Benchmarks: MATLAB

In this second experiment, we embed the Black-Scholes algorithm in form of UDFs into SQL queries.

We again have a HorsePower version, with the Black-Scholes UDF implemented in MATLAB, and a MonetDB version, with the UDF implemented in Python UDF using the NumPy library and the same array programming style as the MATLAB UDF.

In order to understand the implication of having the UDFs written in different programming languages, we first compared the execution time of Black-Scholes written in Python and using HorseIR (both naive and optimized). Execution is in one thread because NumPy does not support multi-threading. Similar to what we have seen with our analysis with MATLAB, a naive usage of HorseIR provides similar execution time as Python (around 500 ms); performing optimizations achieves a speedup of 2.

In order to look at the impact of embedding this UDF into SQL statements, we created both scalar and table UDF variations as well as designed several enclosing SQL statements that offer different potential for optimizations. In particular, 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)
};
```

Furthermore, 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.

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. Furthermore, the selectivity of WHERE clause can be high (returning few records) or low (having many qualifying records).

Table 3 shows the result of all the variations for MonetDB and HorsePower for 1 thread (T1) and 64 threads (T64).

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 first observe that for MonetDB multi-threading has little impact on its performance while HorsePower benefits a lot. As Python is not multi-threaded, the Black-Scholes UDF in MonetDB runs always in a single thread even if 64 threads are enabled, while HorsePower creates optimized parallel also for the Black-Scholes part. But HorsePower is already significantly better with a single thread. We then find that HorsePower has even significant benefits with a single thread. In fact, HorsePower's execution time for the entire query is nearly the same as executing the Black-Scholes algorithm alone, while MonetDB takes nearly double the time (> 900 ms) to execute the entire query than the time used by the Python interpreter to execute Black-Scholes (around 500 ms). 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. As can be seen, 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. MonetDB follows the traditional database optimization technique of applying high selectivity operations first, discarding the records that do not qualify before processing the UDFs. 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

Table 3: **Performance comparison between HorsePower (HP) and MonetDB (MDB) for variations in Black-Scholes.**

UDF	Selectivity	Table UDF (ms)						Scalar UDF (ms)					
		T1			T64			T1			T64		
		MDB	HP	Speedup	MDB	HP	Speedup	MDB	HP	Speedup	MDB	HP	Speedup
bs0_base	100.0%	927.5	249.8	3.71x	774.0	7.09	109x	670.0	249.5	2.69x	696.5	7.06	98.6x
bs1_high	0.2%	926.4	256.2	3.62x	818.0	7.62	107x	6.10	0.32	19.1x	6.55	0.13	50.4x
bs1_low	99.8%	929.7	266.4	3.49x	832.9	14.6	57.0x	725.4	169.6	4.28x	645.4	4.90	132x
bs2_high	0.2%	895.6	4.67	192x	791.5	0.70	1131x	4.29	4.59	0.93x	3.52	0.63	5.59x
bs2_low	99.8%	916.4	11.0	83.7x	820.4	6.64	124x	15.9	10.95	1.45x	5.11	5.95	0.86x
bs3_high	10.0%	911.8	259.0	3.52x	824.4	10.1	81.6x	673.8	179.3	3.76x	623.2	7.69	81.0x
bs3_low	90.0%	879.1	262.5	3.35x	793.6	13.7	57.8x	685.4	182.6	3.75x	641.7	12.8	50.1x

result. A smart system should be able to analyze the semantics of the request and avoid processing the UDF both together. 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. As both avoid executing the UDF, HorsePower has only moderate speedup over MonetDB due to other optimizations. 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. As can be seen, 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. HorsePower has better performance than MonetDB simply because HorsePower can avoid the data movement between the UDF. With more threads, HorsePower's speedup is even better as the data movement in MonetDB is not parallelized and takes most of the time in the whole execution pipeline.

In summary, HorsePower avoids the problems of a black-box integration of programming language execution environments as used in current DBS. As such, it avoids expensive data conversions, can optimize in a holistic manner and provides full support for parallelization, leading to significant speedups.

5 RELATED WORK AND CONCLUSIONS

Intermediate representations and compiler techniques have been applied by others to improve the performance of database queries. However, there is little research in these systems extending to support UDFs within the database queries.

Froid [14] shows a holistic optimization solution by transforming simple UDF to relational code. Thus, the existing query optimizer can be utilized for the optimizations of the execution plan. However, this approach is limited as not all UDFs are translatable to a relational operator.

Weld [12] 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 [10] 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.

In conclusion, `HorsePower` differs from previous work in that it is a compiler-based approach exploiting array-based optimizations to support database queries, MATLAB programs and database queries with analytical UDFs in a holistic framework. Given the very promising evaluation results, future work will integrate different programming languages, and enhance our relational operators.

REFERENCES

[1] [n. d.]. MATLAB. https://www.mathworks.com. ([n. d.]). [Last accessed in Feb. 2021].

[2] [n. d.]. McLab: A Framework for Dynamic Scientific Languages. http://www.sable.mcgill.ca/mclab/. ([n. d.]). [Last accessed in Feb. 2021].

[3] [n. d.]. R Project. https://www.r-project.org. ([n. d.]). [Last accessed in Feb. 2021].

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of Intl. Conference on Parallel Architectures and Compilation Techniques*. 72–81.

[5] 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)*. 37–49.

[6] Hanfeng Chen, Joseph Vinish D'silva, Laurie Hendren, and Bettina Kemme. 2021. *Accelerating Database Queries for Advanced Data Analytics: A New Approach*. Technical Report SABLE-TR-2021-1. Sable Group, McGill University. http://www.sable.mcgill.ca/publications/techreports/#report2021-1

[7] Hanfeng Chen, Alexander Krolik, Bettina Kemme, Clark Verbrugge, and Laurie Hendren. 2020. Improving Database Query Performance with Automatic Fusion. In *Proceedings of Intl. Conference on Compiler Construction, (CC'20)*. 63–73.

[8] 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.

[9] Anton Willy Dubrau and Laurie Jane Hendren. 2012. Taming MATLAB. In *OOPSLA*. 503–522.

[10] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567.

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

[12] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB* 11, 9 (2018), 1002–1015.

[13] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of Intl. Conference on Scientific and Statistical Database Management*. 16:1–16:12.

[14] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB* 11, 4 (2017), 432–444.

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

[16] Transaction Processing Performance Council. 2017. TPC Benchmark H. (2017).