Improving Database Query Performance with Automatic Fusion

Hanfeng Chen
School of Computer Science
McGill University, Canada
hanfeng.chen@mail.mcgill.ca

Alexander Krolik
School of Computer Science
McGill University, Canada
alexander.krolik@mail.mcgill.ca

Bettina Kemme
School of Computer Science
McGill University, Canada
kemme@cs.mcgill.ca

Clark Verbrugge
School of Computer Science
McGill University, Canada
clump@cs.mcgill.ca

Laurie Hendren
School of Computer Science
McGill University, Canada
hendren@cs.mcgill.ca

Abstract

Array-based programming languages have shown significant promise for improving performance of column-based in-memory database systems, allowing elegant representation of query execution plans that are also amenable to standard compiler optimization techniques. Use of loop fusion, however, is not straightforward, due to the complexity of built-in functions for implementing complex database operators. In this work, we apply a compiler approach to optimize SQL query execution plans that are expressed in an array-based intermediate representation. We analyze this code to determine shape properties of the data being processed, and use a subsequent optimization phase to fuse multiple database operators into single, compound operations, reducing the need for separate computation and storage of intermediate values. Experimental results on a range of TPC-H queries show that our fusion technique is effective in generating efficient code, improving query time over a baseline system.

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

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

ACM Reference Format:
Hanfeng Chen, Alexander Krolik, Bettina Kemme, Clark Verbrugge, and Laurie Hendren. 2020. Improving Database Query Performance with Automatic Fusion. In *Proceedings of ACM SIGPLAN 2020 International Conference on Compiler Construction (CC'20).* ACM, New York, NY, USA, **??** pages.

1 Introduction

While database systems have traditionally stored relational tables on a per-row basis, recent developments have shown that a column-store format can outperform traditional RDBMS for many workloads, particularly when the data fits into main memory. One of the first research prototypes of a column-based database manage systems (DBMS) was MonetDB [**?**], still an active research project, but many commercial systems now also support column-based storage (e.g. Microsoft SQL Server [**?**], SAP HANA [**?**], and HyPer [**?**]). This fundamental shift in database architecture has resulted in a large body of research efforts within the database community looking into efficient SQL query execution on such a data format.

Given that in a column store each column of a table is stored as an array, array programming languages appear as a promising construct to implement query execution. In fact, Chen et al. [**?**] introduced an intermediate representation (IR), *HorseIR*, providing efficient database operators. At the same time, it is a general-purpose high-level language, and can therefore serve not only as an effective representation for SQL, but also other higher-level operators and array-based languages such as MATLAB. Thus, compared to existing query compilers that translate from relational algebra to optimized code (e.g. Hyper [**?**]), it is much more general and extensible.

SQL queries are first transformed into execution plans using standard database optimization techniques that describe in which order and how database operators such as joins, selections and aggregations should be executed depending on the data to be analyzed. These execution plans are then translated into HorseIR, the resulting code is optimized using techniques from array programming, and finally the optimized code is compiled into multi-platform C code.

The approach used in HorseIR, however, does not take full advantage of the optimization techniques available. Instead, it is restricted to basic code fusion of element-wise operators, and although it explores some high-level pattern-based optimization techniques, these require considerable knowledge of database operators. Optimization potential is also reduced by extensive use of built-in functions, the complexity of which limits optimization.

For example, it is common for SQL queries to do a selection (selecting some rows based on the filter criteria on one column) and then perform aggregation, such as a summation. In an array-based language this can be represented as

CC'20, February 22–26, 2020, San Diego, CA, USA
2020.

```
R = sum(filter(C, A))
```

where the condition C is a boolean vector, the input array A is a vector with the same number of elements, and the result R is a scalar. A straightforward way to generate C code from this is to generate two loops, the first applying the boolean selection and producing an intermediate result, and the second loop performing the reduction on this intermediate result. A more optimized version, however, would fuse the two loops into one, performing the reduction on the fly and avoiding generation of an intermediate result:

```
R=0;
for(int i=0; i<C.size; i++) {
   if(C[i]) { R += A[i]; } // true block
}
```

The analysis required to generate such code from a series of built-in functions, however, is complex, depending on non-trivial array dependency analysis as well as knowledge of array dimensions, and while possible in principle, our experience has been that these optimization opportunities are not found by our available C compilers.

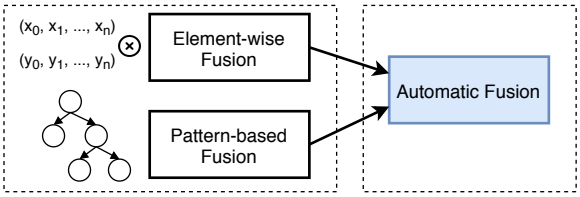

Figure 1. Automatic operator fusion for SQL queries includes element-wise fusion and a small set of patterns.

In this paper we provide a systematic approach to analyze the SQL query programs and generate optimized code. As shown in Fig. **??**, *automatic fusion* is a blend of smart element-wise fusion and a limited set of patterns for cases that cannot be handled. More specifically, we provide a detailed analysis of the most important built-in functions used for executing SQL operators and explain how to optimize across such functions. Our approach is based on a *shape analysis*, which describes the layout and size of variables as functions are applied. This allows us to fuse loops extensively, and avoid unnecessary iterations over the data and intermediate results.

The contributions of this paper are summarized as follows:

- We explore a new approach to improve database query performance by applying techniques derived from array programming. We focus on HorseIR, an IR specifically designed to support database query execution.
- We perform shape analysis on the built-in functions of HorseIR that represent database operators. This allows us to collect precise shape information and provide a conformability analysis to identify fusible sections.
- We present a set of optimization and code generation strategies to automatically generate optimized code.
- We conduct experiments on the database benchmark TPC-H, to show the effectiveness of our technique.

2 Motivation and Background

In this section, we first present the principles of loop fusion techniques for array-based optimization, and then provide background on HorseIR and its optimization capabilities.

2.1 Loop Fusion

Loop fusion is an effective technique to improve program performance as loops usually dominate the execution. In the context of array-based languages, loop fusion is applied during the code generation phase, when array-based functions are translated into loops of the target language such as C. The goal of loop fusion is to minimize the number of loops in the target program. There are two kinds of loop fusion: (1) *Fusion for removing intermediate results*, and (2) *Fusion when sharing the same loop head*.

Ideally, a smart compiler can optimize loops in steps: (I) resolve the data dependency between variables inside loops; (II) identify the use of temporary, intermediate result arrays; and (III) fuse the for loops and avoid the intermediate result arrays whenever possible. However, automatic loop fusion is challenging for compilers [**? ?**]. Instead, we believe it is much easier if the potential of loop fusion is already performed looking at the high-level language (i.e., in the array-based program) due to the higher level of abstraction.

2.2 HorseIR: an Array-based IR for SQL Queries

HorseIR [**?**] was designed to represent execution plans for SQL queries where data is represented in column-format and assumed to reside in main-memory. HorseIR represents the columns of the database tables as vectors, and uses lists for compound data. It provides dozens of data types to accommodate the many data types found in database systems. Moreover, it supports a large set of array-based built-in functions that represent the arithmetic and database-related operators that are commonly used in SQL queries.

Fig. **??** shows a motivating example of queries executing with HorseIR framework. Fig. **??** shows a SQL query on the database table `store_items`. The WHERE clause represents a selection filtering only those records of the table that fulfill certain conditions on the `item_date` column. The SELECT clause projects on columns `item_price` and `item_discount`, performing an element-wise arithmetic function (multiplication) and then aggregation over the result.

Such a query is first translated into a query execution plan. The previous work [**?**] uses the advanced query optimizer of the HyPer database system [**?**] to generate high-performance execution plans. Then, the execution plan is translated into HorseIR using both standard array language and database-specific built-in functions. By not directly translating queries

```
SELECT
  SUM(item_price *
    item_discount) AS saving
FROM
  store_items
WHERE
  item_date >=2010.09.01 AND
  item_date <=2010.09.30;
```

(a) An SQL query example

```
// ... load columns from table
(S0) t0:bool = @geq(c0,2010-09-01:date);
(S1) t1:bool = @leq(c0,2010-09-30:date);
(S2) t2:bool = @and(t0, t1);
(S3) t3:f64  = @compress(t2, c1);
(S4) t4:f64  = @compress(t2, c2);
(S5) t5:f64  = @mul(t3, t4);
(S6) t6:f64  = @sum(t5);
// ... return result as a table
```

(b) Core part of HorseIR code for (a)

```
//... load columns c0,c1,c2
t6 = 0;
for(int i=0; i<n; i++){
  if(c0[i] >= 20100901
     && c0[i] <= 20100930){
    t6 += c1[i] * c2[i];
  }
}
//... return t6, a scalar
```

(c) Optimized C Code generated from (b)

Figure 2. Translating an SQL query from (a) to the corresponding HorseIR code shown in (b) (with columns: c0 (item_date), c1 (item_price), and c2 (item_discount)), and generating optimized C code in (c).

into HorseIR but instead translating the optimized execution plans, the approach leverages the extensive SQL-specific query optimization experience of the database community.

Fig. **??** shows the resulting HorseIR program without optimization and many intermediate results. From there, the previous work [**?**] proposes optimizations based on element-wise fusion and fusion with patterns. For example, (S0) and (S1) in the HorseIR program are easily fused into one statement, as `@gt` and `@lt` are element-wise functions. Since there is a large set of such functions, this approach has shown to be very beneficial. Moreover, generating parallel code for element-wise functions is easy as they are dependence-free.

However, once functions are more complex, such as the `@compress` functions of lines (S3) and (S4) in the example, fusion is no longer straightforward, as these high-level built-in functions contain data-dependent loops implicitly. Thus, fusion across such functions is not supported by the previous work [**?**] which instead generates separate loops for them. This can become a severe performance bottleneck, especially when the intermediate results are barely reused.

The boolean selection function `@compress` is common in SQL queries because of the WHERE clause. In the example, `t2` is a boolean vector indicating the indices of the records that fulfill the filtering condition of the query. In (S3) and (S4) @compress retrieves the corresponding element values of the `item_price` and `item_discount` columns. However, the intermediate results of `t3` and `t4` can be avoided because the retrieval can be fused with the subsequent arithmetic and aggregation functions.

In fact, for the given example, it is possible to fuse everything into one single loop as shown in Fig. **??** (c). In this paper, we show how fusion *across* built-in functions can be done automatically and systematically by using a principled shape and conformability analysis.

Importantly, note that the previous work [**?**], in addition to element-wise fusion, also supports fusion based on pattern rules, such as tree-based patterns. However, using patterns is quite challenging as it requires the expertise of SQL execution plans. Furthermore, there exist many different types of queries, and thus potentially many different patterns. It is not clear, whether the patterns presented in [**?**] are exhaustive.

3 Overall Structure of the Optimizer

We have implemented our approach for generating fused and efficient `C` code in the HorseIR compiler. An overview of the workflow can be found in Fig. **??**.

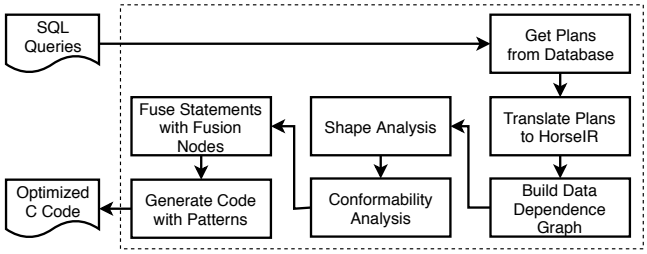

Figure 3. Analysis and code generation overview.

Just as in previous work [**?**], we first generate an optimized query execution plan using the HyPer database system [**?**] [1] from the input query, and translate this plan into HorseIR.

Next, local data-flow analysis processes the intermediate program and computes the shape information at each expression. Shapes are propagated according to rules defined for each built-in function. Using the generated shape information, we then employ conformability analysis to identify fusible sections of code on a data-dependence graph.

Lastly, we optimize the set of fused sections using predefined patterns and generate target C code. Patterns exploit additional optimization opportunities that are frequently present in SQL queries.

[1] We can access HyPer's plan generator online, but HyPer's execution engine is no longer publicly available. See http://hyper-db.de/interface.html

4 Shape Analysis

Shape information is key to identify fusible sections of built-in functions. In this section we: (1) introduce our shape abstraction; (2) categorize built-in functions in groups by shape behaviour; and (3) describe propagation rules for each group.

4.1 Shape Abstraction

Shapes describe the in-memory layout of data. HorseIR has two important shapes used in queries: vectors and lists.

4.1.1 Vector

A vector shape describes a fixed-length one-dimensional array of homogeneous data. It is therefore characterized based on the number of elements as shown in Table **??**.

Table 1. Definitions of vector shapes.

Shape	Description
V(1)	Vector of constant size 1 (i.e. scalar)
V(c)	Vector of constant size c where $c \neq 1$
V(d)	Vector of unknown static size (unique ID d)
V_s(a)	Vector from boolean selection a

The number of elements may either be a compile time constant, or a dynamic value only known at runtime. We include a separate shape for scalar data as some built-in functions exhibit specialized behaviour depending on the exact size.

Dynamic vector shapes describe objects that depend on runtime properties. Our system assigns a unique ID to such vectors, so two vectors with the same ID have the same shape. A specialized dynamic shape, V_s(a), describes the output result of the boolean selection function `@compress` with boolean mask a. The code generator uses this boolean selection shape for further fusion and avoids storing intermediate results.

4.1.2 List

A list shape is composite, storing heterogeneous data in an ordered group of *cells*. Each cell has its own shape, either a vector or a nested list.

```
list_shape ::= { cell_shape };
cell_shape ::= list_shape | vector_shape;
```

We denote a list shape as list<L_0,L_1,...,L_{n-1}>, where L_i is the shape of cell i for n cells. Note that for SQL queries, list cells are always vectors, as they typically represent collections of columns or row indices.

4.2 Built-in Function in Groups

Built-in functions are categorized based on their predefined shape behaviours. This simplifies later analyses which depend on the shape behaviour and not the exact operation. For example, element-wise binary functions `@plus` and `@mul` share identical structure and may therefore share a single set of shape propagation rules.

```
a:i32 = @plus(A, B);
b:i32 = @mul(A, B);
```

HorseIR built-in functions can be categorized into the following groups based on shape behaviour:

Element-wise (E) : unary and binary functions, including arithmetic, boolean, and math. They are frequently used to represent the operators found in the WHERE clause (selection) and the SELECT clause (projection);
Reduction (R) : reduction functions `@sum`, `@avg`, `@min`, and `@max`. Aggregation functions of SQL;
Scan (S) : boolean selection functions `@compress`. After the selection, they retrieve the relevant elements for the projection;
Indexing (X) : indexing function `@index`;
Special Boolean (B) : functions that return a boolean vector without implicit data dependency, such as `@like` and `@member`;
Each (H) : list functions `@each`, `@each_left`, `@each_right`, `@each_item`, and `@raze`. They are often needed in SQL statements with a GROUP BY;
Others (O) : all other functions.

Groups can be extended as needed with additional built-in functions as the language and libraries evolve. Each group is also associated with an abbreviation that is used in the following sections.

4.3 Shape Propagation Rules

Shape propagation rules are defined for each group of vector functions, list functions `@each*`, and other functions.

4.3.1 Vectors

For vector functions, the return shape can either be: (1) a parameter shape; (2) a new vector shape; (3) an error occurs due to a shape mismatch. For case 2, we introduce notation, I, that generates a new dynamic shape. While the new shape may be identical to other shapes at runtime, our static analysis is conservative. Each function group has unique shape rules defined below.

Binary element-wise functions. Binary element-wise functions take two vectors as input, perform an element-wise operation and produce a new vector. If either operand is a scalar, the single value is broadcast. Table **??** presents the shape propagation rules for binary element-wise functions.

Statically known (constant) vector lengths provide exact shape propagation rules and can determine error cases at compile time. If one or more argument is a dynamically known shape, then the resulting shape is also dynamic. If both arguments have the same unique ID or boolean mask, then the argument shape is propagated. In all other cases, a new unique shape is generated.

Unary element-wise functions. Unary element-wise functions take a single vector as input and produce a new output vector of the same size. Dataflow rules for shape are the identity in this case.

Table 2. Rules for binary element-wise Functions (E)

F_B(x,y)	x			
y	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
V(1)	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
V(c_1)	V(c_1)	V(c_0)[1]	I	I
V(d_1)	V(d_1)	I	V(d_0)[2]	I
V$_s$(a_1)	V$_s$(a_1)	I	I	V$_s$(a_0)[3]

[1]: if c_0==c_1 otherwise error
[2]: if d_0==d_1 otherwise I
[3]: if a_0==a_1 otherwise I

Reduction functions. Reduction functions take a vector as input and compute a scalar output value. In all cases this thus produces a $V(1)$ shape.

Table 3. Rules for the scan function (S)

F_S(x,y)	x			
y	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
V(1)	I	error	I	I
V(c_1)	error	I[1]	I	I
V(d_1)	I	I	V$_s$(x)[2]	I
V$_s$(a_1)	I	I	I	V$_s$(x)

[1]: if c_0==c_1 otherwise error
[2]: if d_0==d_1 otherwise I

Scan function. The boolean selection function `@compress` takes two vectors of equal length: a boolean mask vector, and a values vector. The output vector contains only those values with a corresponding TRUE flag in the mask. Table **??** describes the full shape rules where x is the mask and y the values vector.

For constant sized vectors where the length agrees, the output shape is determined at runtime. If the lengths differ a compile-time error is thrown. Dynamic length vectors also generate new scan shapes parameterized on the boolean mask. As multiple value vectors may be compressed using the same mask, we internally map each boolean mask to its output shape. When propagating, this map is first checked before generating a new unique shape. Fig. **??** shows an example of two vectors which have the same output scan shape.

```
// b:bool, x:i32, y:i32 (vectors of same length)
t0:i32 = @compress(b, x);
t1:i32 = @compress(b, y);
// t0 and t1 share the same scan shape
```

Figure 4. Example propagating the scan shape.

Array indexing function. The array indexing function `@index` takes two vectors as input (values and indexes) and performs an indexed read. The output vector therefore contains one element per index, and thus its shape is determined by the shape of the index vector.

Special boolean functions. Special boolean functions take a data vector as input and return a boolean vector indicating adherence to a specified property. For example, `@like(x, y)` checks if the data values `x` match search string `y`. Functions in this group therefore return the shape of the first argument.

4.3.2 Lists

List functions apply a function on cells individually and merge the results into a new list. For example, the `@each` function is shown in Fig. **??**. In the following sections we denote the applied function as `@f`.

```
// x:i32, y:i32 vectors; @f function
t0:list<i32> = @list(x, y);
t1:list<i32> = @each(@f, t0);
// t1 contains cells [@f(x), @f(y)]
```

Figure 5. Example of a list function.

Function each applies function `@f` to each cell in a list. Each cell shape is therefore transformed by the function to create a new list. Given input shape `list<`L_0,...,L_{n-1}`>` the output shape is thus `list<@f(`L_0`),...,@f(`L_{n-1}`)>`.

Function each_left takes two parameters: a list and a variable *of any type*. The function is applied on each cell of the list and the variable to form a new list with cells for each pairing. Given input shapes `list<`L_0, ... , L_{n-1}`>` and `A`, the function produces a new list with shape `list<@f(`L_0`, A),...,@f(`L_{n-1}`, A)>`.

Function each_right takes two parameters: a variable *of any type* and a list. The function is applied on the variable and each cell of the list to form a new list with cells for each pairing. Given input shapes `A` and `list<`L_0, ... , L_{n-1}`>`, the function produces a new list with shape `list<@f(A, `L_0`),...,@f(A, `L_{n-1}`)>`.

Function each_item takes two lists of equal length as input and evaluates the given function on each pair of cells to form a new list. Given input shapes `list<`L_a`>` and `list<`L_b`>` the function returns a new list of shape `list<@f(`L_{a0},L_{b0}`),...,@f(`$L_{a(n-1)}$,$L_{b(n-1)}$`)>`.

Function raze flattens a homogeneous list of vectors into a single vector, removing cell divisions. For any list the output shape is a dynamically sized vector V(d).

4.3.3 Other Functions

For all other functions, a new dynamic shape (either list or vector depending on the return type) is generated as the output shape. This is conservative, but correctly prevents fusing any unknown or non-fusible function. Further optimization is possible using pre-defined patterns.

5 Conformability Analysis

Conformability analysis determines fusible statements of a HorseIR program for code generation. Using the output of shape analysis, we partition the data dependence graph into *fusible sections* and the remaining independent statements. Two statements are in the same fusible section if they are conformable.

5.1 Fusible Sections

A fusible section is a subgraph of the program data dependence graph. Let $G = (V, E)$ represent the data dependence graph with statement nodes and dependence edges. Note that for each statement there is one incoming edge per parameter. The complete graph G can be divided into two parts: fusible (Γ_F) and non-fusible (Γ_N) disjoint subgraphs.

5.2 Conformability

Two statements are conforming if they may be fused in the generated code, thereby eliminating intermediate results. As with the shape analysis, this check is conservative, fusing statements only if provably correct. Trivially, element-wise functions operating on the same vector shape may be fused, but we can also fuse both boolean selection and reductions. The basic rules for conformability are described in Table **??**. In our approach, we check conformability between statements and their *definition statements* of the input parameters (predecessors in the data dependence graph).

Table 4. Conformable rules for two shapes

	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
V(1)	✓	×	×	×
V(c_1)	×	c_0==c_1	×	cond(a_0,c_1)
V(d_1)	×	×	d_0==d_1	cond(a_0,d_1)
V$_s$(a_1)	×	cond(a_1,c_0)	cond(a_1,d_0)	a_0==a_1
cond(a,y) is ✓ if a.size == y else ×				

5.2.1 Algorithm

Conformability analysis produces a list of fusible sections given the conformability of the statements. It traverses bottom up on the dependency graph (reverse topological order), and recursively fuses definition statements that are conformable with their uses. Each recursive call tree therefore forms a single fused section that ends when no more statements may be fused. In addition to conformability, we ensure that reductions may only terminate fused sections and not be internal nodes. This restriction is due to the synchronization and implicit data-dependence introduced by the reduction behaviour. Our algorithm for vector fusion is described in detail in Algo. **??** and subsequent sections.

Algorithm 1: Finding Fusible Sections for Vectors.

```
Input: Data dependence graph G
Output: A list of fusible sections
let ∅ be an empty vector;
allStmts ← reversed topological order of the graph G;
foreach stmt A in allStmts do
    if isNotVisited(A) then
        if getOp(A) is a reduction function then
            section ← findFromReduction(A);
        else
            section ← findFusibleSection(A);
Function findFusibleSection(A):
    if isNotVisited(A) then
        setVisited(A);
        if isGroupE_Binary(A) or isGroupS(A) then
            list ← fetchFusibleStmts(A, A.first.parent);
            list.append(fetchFusibleStmts(A,
              A.second.parent));
        else if isGroupE_Unary(A) or isGroupB(A) then
            list ← fetchFusibleStmts(A, A.first.parent);
        else if isGroupX(A) then
            list ← fetchFusibleStmts(A, A.second.parent);
        else
            list ← ∅;
        return {A}.append(list)
    return ∅;
Function fetchFusibleStmts((A,P)):
    if isConformable(A, P) then /* Rule 2 */
        return findFusibleSection(P);
    return ∅;
Function findFromReduction(A):
    setVisited(A);
    return {A}.append(findFusibleSection(A.first.parent));
```

5.2.2 Vector Conformability

Identifying fusible sections for vector functions is divided into two passes. The first pass identifies the main fusible sections, while the second pass corrects any data dependencies between sections. The algorithm terminates when all statements have been visited.

1st pass: Finding all eligible statements for a fusible section verifies for type and shape conformability.

Rule 1: candidate statements need concrete types (no wildcard or unknown types) and have built-in functions belonging to groups {`E,R,S,X,B`}.

Rule 2: candidate statements must be conformable with the shape of the definition statements according to Table **??**.

Each iteration of the algorithm identifies statements adhering to the first rule, and recursively checks definition statements for both rules 1 and 2 as seen in function `findFusibleStmts`. If the definition statement contains a reduction, a new fusible section is started and processed in the function `findFromReduction`.

The function `findFusibleSection` traverses the built-in functions according to their group: (1) traversing the parents of both parameters for binary element-wise functions `E` and the scan function `S`; (2) traversing the parent of the first parameter for unary element-wise functions `E` and special boolean functions `B`; and (3) traversing the parent of the second parameter for indexing functions `X`. Other functions leave the list of fusible sections unchanged.

2nd pass: Trimming sections that introduce dependencies.

The algorithm described in Algo. **??** optimistically creates fusible sections, assuming that intermediate results are not required for other computations. If a definition is used in more than one successor and the successors are partitioned into separate fused sections, a data dependence will exist between sections. This dependency would require an intermediate result be stored, which negates the purpose of our approach. We therefore remove any statement whose successors are in different fused sections from the fused section.

5.2.3 List Conformability

A fusible section of list-shaped code ends with the pairing of a list reduction (e.g. `@each(@sum,...)`) and `@raze`. This combination produces a vector with a single value per list cell. We then recursively expand the section checking conformability between the current statement and predecessor `@each*` calls as done for vectors. We additionally impose that the applied function in the `@each*` calls has appropriate shape behaviour: `@each_left` requires either group `B` or `E`, `@each_right` requires group `X` or `E`, and `@each_item` only group `E`. Boolean selection functions (`S`) are not supported.

5.3 An Example

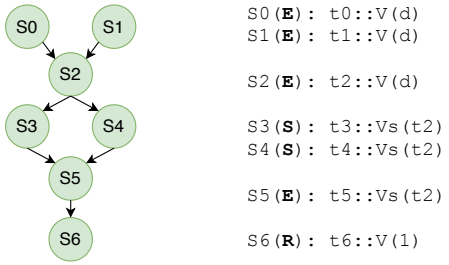

Figure 6. A fusible section for the HorseIR program in Fig. **??**. The text format on the right hand side is `<statement>(<group>): <variable>::<shape>`.

Given the HorseIR program in Fig. **??**, our algorithm identifies a fusible section shown in Fig. **??**. Initially, the variable `c0` is assigned a dynamic vector shape with unique ID, `V(d)`, as the exact size depends on the input table. Next, the 3 element-wise functions in statements `S0`, `S1`, and `S2` propagate the vector shape `V(d)` according to the shape rules. Both compression functions in `S3` and `S4` generate a new shared scan shape `Vs(t2)` as they both use the same boolean mask. The following binary function uses this equality to correctly infer its output shape. Finally, the reduction function `@sum` returns a vector with one element. Note that all functions in the computed fusible section share the same loop range, `V(d)`.

The code in Fig. **??** shows the generated C code for the complete fusible section. Note the presence of an internal condition for the boolean selection and the accumulator for the reduction. Details of the code generation strategy for fusible sections are found in Sec. **??**.

6 Code Generation Strategies

Our code generation strategy follows a pattern based approach for fused sections, whereas normal (non-fused) code calls optimized library functions. First, each fused section is associated with a fusion node, the nodes are optimized, and lastly the code is emitted.

6.1 Fusion Nodes

Each fused section is associated with a fusion node, a collection of metadata used for generating the loop. For each section, we traverse the statements and collect: (1) loop bounds; (2) fused expressions; (3) boolean mask (if any); and (4) reduction operation (if any). The set of properties determines which code generation pattern is used.

6.2 Code Generation for Vectors

```
// reduction: YES
// scan: YES
for(i=0; i<len; i++){
  if(cond[i]){
     z = z Rop expr_rhs;
  }
}
z = Rfinal(z);
```

(a) Case 0

```
// reduction: YES
// scan: NO
for(i=0; i<len; i++){

  z = z Rop expr_rhs;

}
z = Rfinal(z);
```

(b) Case 1

```
// reduction: NO
// scan: YES
c = 0;
for(i=0; i<len; i++){
  if(cond[i]){
    z[c++] = expr_rhs;
  }
}
```

(c) Case 2

```
// reduction: NO
// scan: NO

for(i=0; i<len; i++){

  z[i] = expr_rhs;

}
```

(d) Case 3

Figure 7. Code generation for vectors. `Rop`: reduction operation; `Rfinal`: final reduction step (e.g. divide by element count); `z`: accumulator/output vector.

Code generation for fused vector operations follows 4 patterns depending on the presence of reduction and boolean selection. Each iterates over the length of the list, fuses the RHS expressions, and produces the appropriate output. Reduction nodes accumulate a scalar value, while non-reduction nodes create a new vector. `Rfinal` performs the final step of the reduction (e.g. dividing by the number of elements to compute an average). In the case of compression, the condition is first evaluated and the RHS computed if necessary. Fig. **??** shows the variations of the code generation patterns.

Note that when generating parallel code for Fig. **??**, we employ a strategy that: (1) counts the number of true elements in each segment and computes an offset for each segment; and (2) divides the boolean vector into segments evenly based on the number of cores. Each thread thus maintains a segment of the boolean vector independently. For all other cases, typical parallel strategies are effective.

6.3 Generating Code for Lists

```
for(i=0; i<list.len; i++){ /* loop over cells */
  cell = list[i]; /* fetch one cell */
  /* init t */
  for(j=0; j<cell.len; j++){ /* loop over content */
     t = t Rop (cell[j])
  }
  z[i] = Rfinal(t); /* store final value */
}
```

Figure 8. Code generation for lists. `Rop`: reduction operation; `Rfinal`: final reduction step (e.g. divide by element count); `t`: cell accumulator; `z`: output vector.

List fusion nodes compute a single value per list cell and return a vector. Fig. **??** shows the code generation pattern for lists. As seen in the figure, there are two loops present: an outer loop iterating over cells and an inner loop computing the reduction expression for each cell.

Note that the ratio of `list.len` and `cell.len` may vary greatly. When parallel code is generated, we may therefore parallelize the outer or inner loop depending on the data. In our implementation, we use a simple runtime heuristic based on the size ratio to choose which loop runs in parallel.

6.4 Further Fusion Opportunities

Fusible sections can be further fused if: (1) they share the same loop head, and (2) there is no data dependence between the loop bodies. This is particularly useful and common in column-based IMDBs where data is fetched from multiple independent columns using a single array of indices (e.g. the result of a join). It also improves parallelization as the number of synchronization barriers is reduced.

7 Evaluation

In this section we evaluate the performance of our optimizations by conducting experiments on the TPC-H benchmark.

7.1 Methodology

Experimental setup. We run all benchmarks under Ubuntu 16.04.6 LTS on a server which has 4 Intel Xeon E7-4850 2.00GHz (total 40 cores/80 threads) and 128 GB RAM. We use GCC compiler with the version *v8.1.0* to compile the generated C code with optimization level `-O3` and `-march=native` enabled. We use the latest MonetDB [**?**] released in *Apr2019* with version v11.33.3, as a baseline comparison for the following HorseIR versions: (1) HorseIR-noopt: no optimizations; (2) HorseIR-opt1 : element-wise and pattern-based fusion from previous work [**?**]; and (3) HorseIR-opt2 : fusion approach presented in this paper.

Execution time. Our results present the core execution time of the database query. That is, compilation time, input data loading time and results output time are not considered as we want to zoom in on the effects of the optimization. The results present the average over 15 executions for each query.

TPC-H SQL benchmarks. TPC-H [**?**] is a widely used SQL benchmark suite for analytical data processing simulating real Business to Consumer (B2C) database applications. The database has 8 tables over which 22 queries are defined. The database size can be varied by indicating a *scale factor* (SF). For example, a scale factor of 1 (i.e. SF1) means 1GB of input data. With an increasing scale factor, (nearly) each of the tables holds more data records. Our results are for SF1 but initial results on larger scale factors show similar results.

For this paper, we have selected the subset of queries in which the basic built-in functions that we aim to fuse have a major impact on execution time. In particular, these queries have a maximum of 2 joins. Joins are very expensive, and in queries with more than 2 joins, the join execution takes up most of the time, thus, the optimizations presented in [**?**] and in this paper have less impact. Hence, we have such 8 queries (q1/4/6/12/14/16/19/22) in our experiments. Future work will look more closely at fusion potential for join operations.

7.2 Execution Time Results

Fig. **??** shows the execution times using MonetDB, HorseIR-noopt, HorseIR-opt1, and HorseIR-opt2 on SF1 with increasing number of threads. Execution time generally decreases with increasing number of threads up to a certain threshold. The sweet spot for both MonetDB and HorseIR, where the best performance is achieved, is around 16 threads. Thus, using parallel execution is beneficial for most queries. In q16, however, there are many small cells (18314 cells, average size 6.5) and thus our vector parallelization is underutilized.

HorseIR-noopt has the worst performance in most cases showing that optimization techniques are crucial when trying to exploit array-based languages for query execution.

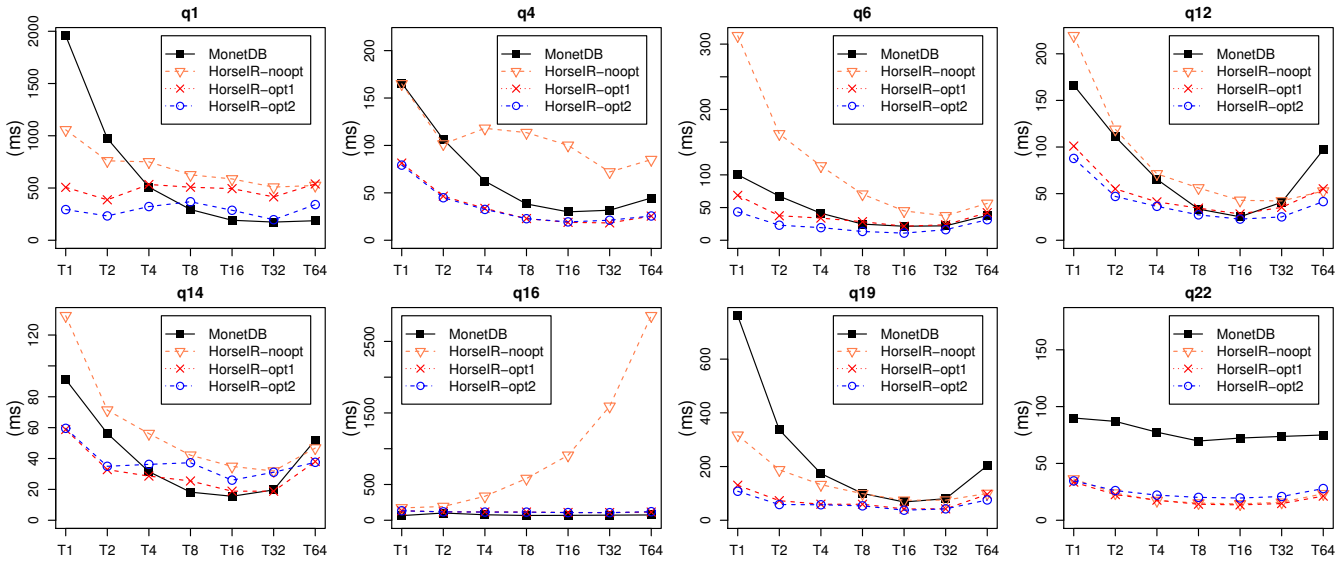

Figure 9. The result of TPC-H queries with 1GB input data (SF1).

The two HorseIR-opt versions are generally not as sensitive to the number of threads as MonetDB, which shows quite bad performance for many queries when there are only a few threads. When looking only at the optimal thread level around 16, MonetDB shows the best performance for one query, HorseIR-opt1 for one query, HorseIR-opt2 for two queries, HorseIR-opt1 and HorseIR-opt2 for two queries, and all three behave very similarly for two queries. That is, there are only 2 queries where our approach is worse than one of the other approaches, and only by a small margin.

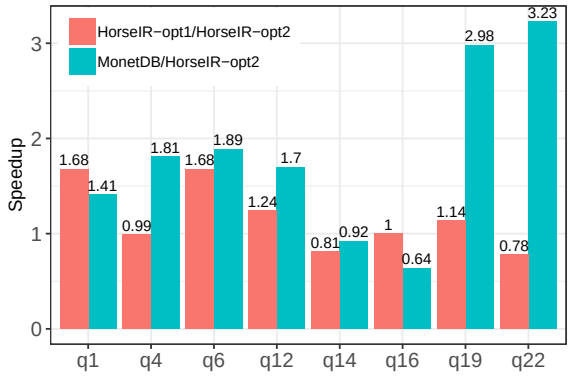

Figure 10. Geometric means of TPC-H queries with 1GB input data (SF1).

To better understand the performance benefit of our approach across all thread counts, Fig. ?? presents the geometric mean speedup for HorseIR-opt2 over HorseIR-opt1 and MonetDB. HorseIR-opt2 provides an improvement over MonetDB (speedup > 1) for all but one query and the improvement is quite significant.

When we are better. In q1, our optimizer identifies eight fusible sections that are merged into one big loop. In q6, two blocks of element-wise functions separated by a non-element-wise statement (`@compress`) are fused together. In both cases, intermediate results are avoided and performance improved. In q12/q19, more statements are fused and less intermediate results created but the effect is weaker.

When we are worse. In q14/q22, the fused loop sizes are relatively small with long expressions in the body. Thus, fusion introduces complexity without significantly fewer iterations.

When we behave similarly. For queries q4 and q16, the filters are very selective and only a few rows qualify. Thus, fusing loops has little benefit but also does not harm the execution.

Compared to HorseIR-opt1, we improve four queries, are the same for two queries, and are worse for two queries giving a geometric speedup of 12%. This performance improvement is due to the significant increase in fused statements discussed in Sec. ??. Our technique thus effectively identifies fusion opportunities in HorseIR programs generated from SQL queries using a more systematic and general approach than patterns.

7.3 Fusing Statements

Fig. ?? shows the the number of fused statements using element-wise fusion in HorseIR-opt1 and the more general function fusion in HorseIR-opt2. Pattern-based fusion is not shown as both approaches use the same patterns.

As shown, our approach always fuses more statements than HorseIR-opt1. In q1 and q22, the large increase in fused statements is due to list-related statements that can now be fused without fixed patterns. Furthermore, our optimizer exploits fusion of mixed function kinds such as element-wise and reduction functions. However, as seen in the previous

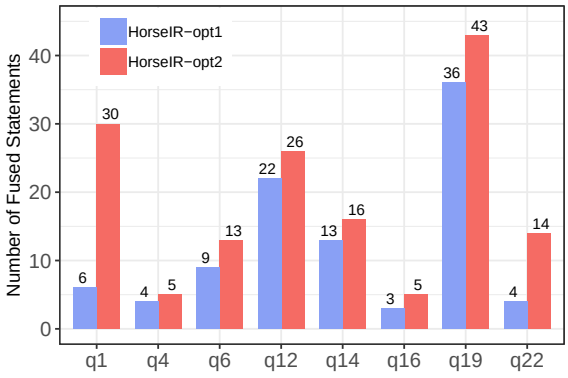

Figure 11. Number of element-wise fused statements in HorseIR-opt1 and our new fusion in HorseIR-opt2.

analysis, fusing is not always beneficial. In q22, we fuse significantly more than HorseIR-opt1, but the loops now contain complex expressions which result in worse performance.

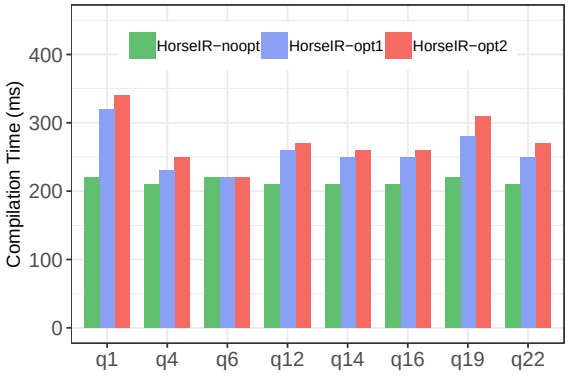

Figure 12. Compilation time for generated C code.

The compilation time of the generated C code seen in Fig. **??** shows that our technique is slower to generate the binary than more naive approaches. However, the increase in compilation time is associated with exploitation of additional optimization opportunities that benefit the execution time. With large data sizes common to modern applications or repeated query execution, the compilation time can be effectively amortized.

7.4 Discussion

In summary, fusion across statements is beneficial in many cases. However, there are two situations in which to be careful applying fusion. Firstly, fusing too many statements might become sub-optimal, likely due to the increased register pressure. In practice, one could create heuristics to determine a maximum number of statements to be fused.

The second situation arises when filtering conditions have a high selectivity, e.g., when only 10 out of a million records qualify. In this case, the benefit of avoiding intermediate results is negligible, while the overhead of code fusion might become a factor. If one had enough information about the actual data, such unnecessary fusions could be avoided. Therefore, introducing runtime optimizations deciding when to fuse is an interesting avenue for future research.

8 Related Work

Fusion techniques are popular due to their success in reducing intermediate results. This is especially true for compiled environments which easily allow code fusion.

Fusion in programming languages

In the MATLAB-to-FORTRAN compiler [**?**], shape and size information can be obtained from a conformability analysis with a set of well-defined conformable operators for scalar, vector, and matrix. However, no further operator fusions after conformability analysis are provided.

In the R programming language, a vectorizer is proposed for the built-in function `Apply` [**?**]. The function `Apply` is similar to our list-based functions, which take a function and a list of data inputs as parameters and repeatedly apply the function. Their vectorizer involves both code and data transformation while we focus on code optimization. Additionally, we explore a wider set of built-in functions rather than only a single list-based function.

Ju et al. [**?**] investigate built-in function fusion in a pure array-based programming language, APL. They classify operators into groups based on the features of the functions and present a model to help generate parallel code when fusing built-in functions. Ching et al. [**?**] use simple techniques to fuse arithmetic functions when compiling APL to parallel C code. In contrast, our work is aimed at array-based programs generated from SQL queries, considering functions important to database operators.

Similar to optimizations for arrays [**? ?**], we exploit type and shape information of arrays to generate efficient code. But due to the high-level semantics of HorseIR programs, we avoid complicated vectorization techniques [**? ?**].

Fusion in database query optimizations

Loop fusion for query programs is typically achieved through a fixed set of complex predefined set of rules. The DBLAB/LB query compiler [**?**] provides fusion rules for different loop fusion algorithms to generate optimized code [**?**]. WeldIR [**?**] adopts rule-based optimizations for element-wise and common-loop-head fusion. HorseQC[**?**] and HIQUE [**?**] present rule-based fusion strategies. However, just as the patterns in [**?**], the rule-based approach is challenging to generalize.

HyPer [**?**], an in-memory database system, adopts a data-centric model when compiling SQL queries to LLVM code. They developed a greedy algorithm to produce/consume operators directly in an execution plan and fuse relational operations directly. Peloton [**?**] considers operator fusion for operators within a pipeline. The stream-fusion [**?**] needs to define extra fusion-related constructs for collections.

By contrast, our fusion strategy is a systematic approach which collects precise shape information on a well-defined array-based language for automatic fusion.

Performance-oriented systems

TVM [**?**], a system designed for deep learning, introduces operator fusion for graph operators when generating efficient GPU code. Their approach is limited, as the fusion rules are fairly simple and specific to fusion between groups. We provide a more sophisticated fusion approach which considers more groups and defines systematic data-flow analysis to identify fusion opportunities.

LIFT [**?**] shows a different approach by defining complex functional patterns with precise descriptions for generating efficient code for parallel devices such as FPGA, as its input code from various domains rather than database queries.

9 Conclusions and Future Work

Array-based languages are a natural fit for column-oriented approaches to database systems. With a straightforward code generation strategy, however, benefits are reduced by the need to store intermediate computations, and this is not easily compensated for by optimizations normally applied in the low-level, generated code. Our work exploits a higher level intermediate representation (HorseIR) to demonstrate that a well-informed, methodical optimization approach can help identify loop fusion opportunities, allowing the multiple steps implicit in query execution to be efficiently aggregated into single loops with composite operations. This greatly improves performance over more naive query-code generation, both in single and multi-processor execution contexts.

Future work is aimed at expanding the set of strategies we have for fusing array-based operations. Much of our current design remains conservative, and dynamic specialization or more complex, in-loop control flow would potentially allow fusing statements with different shapes. We are also interested in optimizing compile time, which would enable our approach to be more easily applied to runtime query generation.