Indian Institute of Technology, Kanpur

Department of Computer Science and Engineering

Project Report

CS738: Advanced Compiler Optimizations

Academic Year 2020 - 2021

Ankita Dey (20111013)
ankitadey20@iitk.ac.in

Tamal Deep Maity(20111068)
tamalmaity20@iitk.ac.in

Tanisha Rastogi (20111069)
tanisharas20@iitk.ac.in

1 Abstract

Research has shown that RDBMS implementations optimized for large scale data processing tend to benefit from a column-store architecture, an approach where each column is stored contiguously on its own . Here a column's logical data structure is akin to that of a programming language array.

Array based languages have good column based query handling techniques on which optimization techniques work effectively. Several SQL queries can be typically merged in this process to a single query and hence this proves to be very effective.

2 Introduction

We have tried to reimplement the paper 'Improving database query performance with automatic fusion' (link: https://dl.acm.org/doi/abs/10.1145/3377555.3377892) by Hanfeng Chen, Alexander Krolik, Bettina Kemme, Clark Verbrugge, and Laurie Hendren. Henceforth any refernce to it will be refered to as the paper.

We think this project is relevant to CS738A since we have used a data dependence graph generated from an array based Intermediate Representation called HorseIR to optimize SQL query and generate fused and optimised C code.

3 Overview of Paper

In the paper, they have first generated an optimized query execution plan using the HyPer database system from the input query, and translated this plan into HorseIR. Next, local data-flow analysis processes the intermediate program and creates data dependent graph and from it computes the shape and function information.

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

Figure 1: Definition of Shapes in HorseIR

Using these information, conformability analysis is employed to identify fusible sections of code on a data-dependence graph. Lastly, the set of fused sections is used to generate target C code.

	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 ×				

Figure 2: Conforming rules for two shapes in HorseIR

4 Implementation

4.1 Overview

Since the HyPer's execution engine is no longer publicly available (also declared in the paper itself) and we do not have access to HorseIR, we have translated the SQL query to a C++ program. We tried to keep the unoptimized C++ program and the input query as equivalent as possible. Next we tried to implement all the steps according to our understanding of the paper to obtain the optimized C++ program as our final output as given in the paper.

4.2 Translating SQL query to C++ code

```
1 SELECT
2   SUM(item_price *
      item_discount) AS saving
3 FROM
4   store_items
5 WHERE
6   item_date >=2010.09.01 AND
7   item_date <=2010.09.30;
```

Figure 3: SQL query

Taking an idea from the example SQL query given in paper, we have created a table with M rows , where M is a very large number (we considered M = 1000000) and 3 columns in C++ . The 1st column denotes the Item Price, 2nd column denotes Item Discount and 3rd column denotes Item Date. We have filled up the rows using dummy values.

```
// ... 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
```

Figure 4: Core part of HorseIR code from SQL Query of Figure 3

Next according to the HorseIR expected to be generated upon translating the given SQL query in figure 3, each of the statements S0 to S6 has been implemented in the C++ code.

S0 and S1 has been implemented by generating a table with values greater than or equal to and less than or equal to constants, 33% of M and 66% of M respectively. S2 basically looks like inner join so for S2, tables generated for S0 and S1 have been traversed and those values present in both the tables are stored in a new table. For S3 and S4 respectively,

column 1 and column 2 of output of S2 have been appended. For S5 output of S3 and S4 have been multiplied and finally for S6, all elements of output table of S5 have been added.

4.3 Shape Analysis and Conformability Analysis

Firstly a data dependence graph is created from the HorseIR given in 4. We have used string manipulation to process each statement in the HorseIR segment and obtained the data dependence graph shown in Figure 5

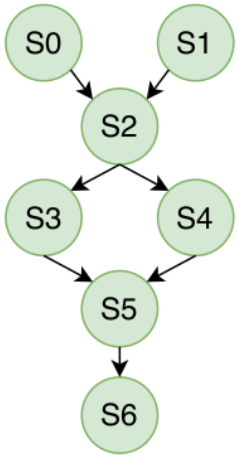


Figure 5: Data dependence graph formed from HorseIR of 4

Next reversed topological ordering of the graph G is done. Reverse topological ordering of the data dependence graph gives a linear ordering such that statements that have argument(s) dependent on result of some other statements will be ordered before the later. We have mapped the output of statements with their shape and the function used in each statement with the type of function as per the paper. Parents of the nodes mentioned in the algorithm have been obtained from the Data Dependence graph. The shapes and the functions determine if two statements are fusible or not. To sum it up, we have tried to implement the algorithm given in next page (Figure 7) to get the list of fusible sections.

Finally we get the following Fusible sections as shown in Figure 6

```
S0(E): t0::V(d)
S1(E): t1::V(d)

S2(E): t2::V(d)

S3(S): t3::Vs(t2)
S4(S): t4::Vs(t2)

S5(E): t5::Vs(t2)

S6(R): t6::V(1)
```

Figure 6: Fusible sections of HorseIR

```
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 isConforming(A, P) then /* Rule 2 */
        return findFusibleSection(P);
    return ∅;
Function findFromReduction(A):
    setVisited(A);
    return {A}.append(findFusibleSection(A.first.parent));
```

Figure 7: Algorithm to find Fusible section

The functions of HorseIR referred to in this algorithm while checking Group are:

Element-wise (E) : refers to unary and binary functions used to represent the operators found in the WHERE clause (selection) and the SELECT clause (projection).

Reduction (R) : refers to @sum, @avg, @min and @max, i.e. 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.

4.4 Optimized Code Generation

```
1 //... load columns c0,c1,c2
2 t6 = 0;
3 for(int i=0; i<n; i++){
4   if(c0[i] >= 20100901
5      && c0[i] <= 20100930){
6     t6 += c1[i] * c2[i];
7   }
8 }
9 //... return t6, a scalar
```

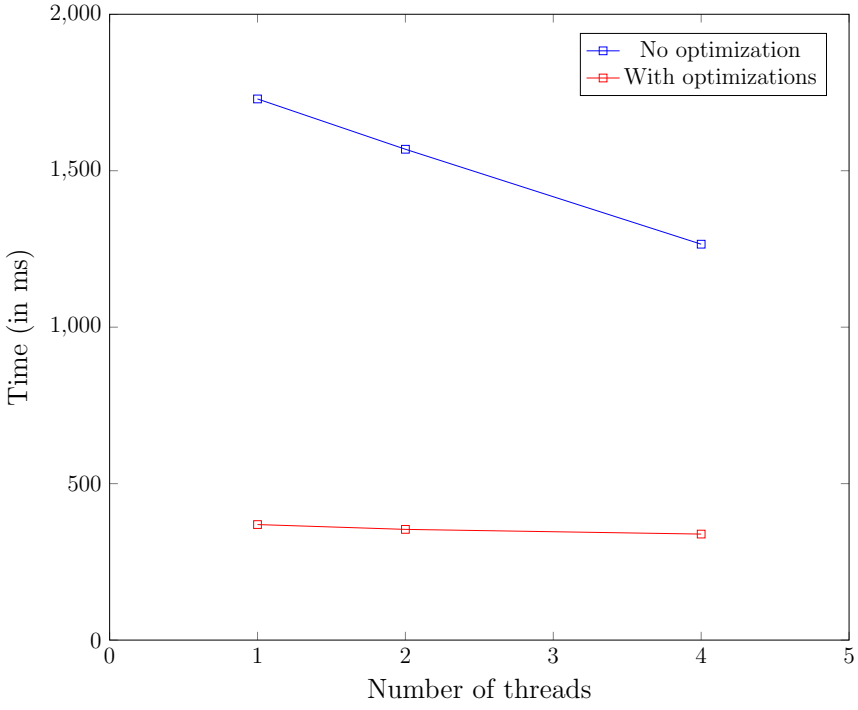
Figure 8: Optimized C++ Code generated from Figure 4 query

After finding the fusible sections as shown in Figure 6, we have written the optimized C++ code on our own as shown in Figure 8. The code generates the same table with 3 columns and M rows as done for the unoptimized C++ code (3).

We choose C++ as the language to write our optimized code in because the query execution time would run on the same platform as the sql query is written in.

5 Evaluation and Conclusion

On running the unoptimized C++ code and the final optimized C++ code we could see the following difference in time taken (average of 5 runs):



By translating SQL execution plans to HorseIR, many standard compiler optimizations and parallelizations can be applied to HorseIR which leads to improved performance for CPU-bound and memory-intensive database queries.