HorsePower: An Array-based Optimization Framework for Query Processing and Data Analytics

by

Hanfeng Chen

School of Computer Science

McGill University, Montreal

February 2021

A thesis submitted to McGill University

in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Abstract

Relational database management systems (RDBMS) are operationally similar to a dynamic language processor. They take SQL queries as input, dynamically generate an optimized execution plan, and then execute it. In recent decades, the emergence of in-memory databases, which use array-like storage structures to store the columns of the database tables, shifted the focus towards CPU-bound query optimizations. The similarity in the computational characteristics of such database workloads and array programming language optimizations have so far been largely unexplored. We believe that these database implementations can benefit from merging database optimization techniques with dynamic array-based programming language approaches; even more so, as database queries are more and more used together with analytics functions in data science workflows. Therefore, we present HorsePower, a framework for optimizing both database queries and supporting advanced data analytics. The framework employs a newly designed array-based intermediate representation, HorseIR, that resides between database queries and programs written in high-level languages on the one side, and compiled code on the other side. It provides translators to generate HorseIR code from database execution plans and programs written in the popular programming language MATLAB, an interpreter to execute HorseIR, and a compiler that optimizes HorseIR and generates efficient CPU and GPU code. We compare HorseIR with a relational database management system MonetDB and MATLAB, by testing standard SQL queries, SQL queries with embedded analytics functions written in MATLAB, and MATLAB benchmarks to show how our approach and compiler optimizations improve the runtime of complex queries and analytics functions.

Résumé

Les systèmes de gestion de base de données relationnelle (SGBDR) sont fonctionnellement similaires à un processeur de langage dynamique. Ils prennent les requêtes SQL en entrée, génèrent dynamiquement un plan d'exécution optimisé, puis l'exécutent. Au cours des dernières décennies, l'émergence de bases de données en mémoire, qui utilisent des structures de stockage de type tableau pour stocker les colonnes des tables de base de données, a déplacé l'attention vers l'optimisation des requêtes liées au processeur. La similitude des caractéristiques de calcul de ces charges de travail de base de données et des optimisations du langage de programmation de tableaux a jusqu'à présent été largement inexplorée. Nous pensons que ces implémentations de base de données peuvent bénéficier de la fusion des techniques d'optimisation de base de données avec des approches de langage de programmation dynamique basées sur des tableaux ; d'autant plus que les requêtes de base de données sont de plus en plus utilisées avec des fonctions d'analyse dans les workflows de science des données. Par conséquent, nous présentons HorsePower, un cadre permettant d'optimiser les requêtes de base de données et de prendre en charge l'analyse de données avancée. Le framework utilise une représentation intermédiaire basée sur un tableau nouvellement conçue, HorseIR, qui réside entre les requêtes de base de données et les programmes écrits dans des langages de haut niveau d'un côté et le code compilé de l'autre côté. Il fournit des traducteurs pour générer du code HorseIR à partir de plans d'exécution de base de données et de programmes écrits dans le langage de programmation populaire MATLAB, un interpréteur pour exécuter HorseIR et un compilateur qui optimise HorseIR et génère un code CPU et GPU efficace. Nous comparons HorseIR avec un système de gestion de base de données relationnelle MonetDB et MATLAB,

en testant des requêtes SQL standard, des requêtes SQL avec des fonctions d'analyse intégrées écrites en MATLAB et des benchmarks MATLAB pour montrer comment notre approche et les optimisations du compilateur améliorent le temps d'exécution des requêtes complexes et des fonctions d'analyse.

Acknowledgements

My first and special thanks give to my dear advisors Laurie J. Hendren and Bettina Kemme. Completing my Ph.D. program in research would have been impossible without their consistent encouragement and support. Laurie J. Hendren was a prominent professor working on the cutting-edge research field of compiler tools. She was also a keen person exercising her hobbies in music and sports. She inspired me to continue exploring academic research and maintain an optimistic life attitude. I still remember the day in 2014 I was invited to give a presentation in her lab. Later, it turned out that I started my Ph.D. under her supervision in the following year. Bettina Kemme is an incredible professor supervising me after I decided to extend my research interests from programming languages to database query processing. I appreciate her valuable feedback and guidance that broadened my understanding in database research.

My second thanks go to people who used to collaborate with me directly in research. In particular, I would like to thank Joseph Vinish D'silva, an excellent teammate for sharing his expertise in database research; Alexander Krolik, a great programmer and organizer for his contribution in clarifying the design and implementation issues of my research project; and Clark Verbrugge for his free walk-in office hours for research discussions. I also thank other people: David Herrera, Erick Lavoie, and Hongji Chen for their hard work in our paper collaborations.

My third thanks go to my labmates and friends. I remember that Prabhjot Sandhu, another Ph.D. student, and I went to yoga classes in the university gym for relaxing after intensive study in our first year. Other than that, I have other common memory for activities shared with Erick Lavoie, Joseph Vinish D'silva, Prabhjot

Sandhu, Alexander Krolik, David Herrera, Amir Bawab, Vincent Foley-Bourgon, Akshay Gopalakrishnan, Duan Li, Zhening Zhang, Faiz Khan, Steven Thephsourinthone, Yulan Feng, Hongji Chen, Yi Chang, and Yu Wang. It is definitely important to have had all of you around for my study and life in Montreal in the past years.

My last thanks go to my lifelong mentor Wai-Mee Ching and my family. Wai-Mee Ching played a vital role in my decision to pursue a Ph.D. degree after completing my master. It was pleasant to discuss research ideas with him and received his insightful feedback. Despite the fact that my parents and my sister have no idea about my research, they fully supported me in the past years. Without their continuous mental support, the research in this thesis could have not been carried out.

Contents

Abstract **i**

Résumé **iii**

Acknowledgements **v**

Contents **vii**

List of Figures **xiii**

List of Tables **xvii**

List of Abbreviations **xix**

1 Introduction **1**

1.1 Objectives 3

1.2 Approach Overview 3

1.3 Contributions 5

1.4 Publications 6

1.5 Thesis Organization 8

2 Background **9**

2.1 Query Processing in Relational Database Systems 9

2.1.1 Relational Model 10

2.1.2 Database Storage 11

2.1.3 Relational Algebra and SQL . . . 11
2.1.4 Database Query Processing . . . 14
2.1.5 User-defined Functions in Database Queries . . . 15
2.2 Array Programming Languages . . . 19
2.3 Intermediate Representations . . . 23

3 Outline of HorsePower 25
3.1 HorseIR Design . . . 26
3.2 Front-end Design . . . 27
3.3 Back-end Design . . . 27
3.4 Built-in Functions . . . 29
3.5 Runtime Support . . . 29

4 HorseIR: the Core 31
4.1 Introduction and Design Principles . . . 31
4.2 Program Structure . . . 32
4.2.1 Modules . . . 34
4.2.2 Methods . . . 34
4.2.3 Blocks and Scoping . . . 34
4.3 Types . . . 36
4.3.1 Base Types and Homogeneous Arrays . . . 38
4.3.2 Advanced Heterogeneous Data Structures . . . 39
4.3.3 A special type: wild-card . . . 43
4.4 Functions . . . 44
4.4.1 Vector-based Functions . . . 45
4.4.2 List-based Functions . . . 45
4.4.3 Database-related Functions . . . 46
4.4.4 Auxiliary Functions . . . 46
4.5 Program Statements . . . 46
4.5.1 Expression Statements . . . 47
4.5.2 Assignment Statements . . . 47

4.5.3 Control Statements 47

5 Front-end: Compiling to HorseIR **51**

5.1 HorseSQL: SQL-to-HorseIR Translator 52

5.1.1 Mapping Relational Algebra to HorseIR 52

5.1.1.1 Projection 52

5.1.1.2 Selection 53

5.1.1.3 Join 54

5.1.1.4 Aggregation 57

5.1.1.5 Group By 57

5.1.1.6 Order By 59

5.1.2 Code Generation Strategy 60

5.1.3 Optimizations in Generating HorseIR Code 63

5.2 HorseMATLAB: MATLAB-to-HorseIR Translator 68

5.2.1 Mapping MATLAB to HorseIR 68

5.2.2 Example Code 71

5.3 HorseUDF: UDF-to-HorseIR Translator 74

6 Back-end: Execution on HorseIR **77**

6.1 HorseInterpreter 78

6.2 HorseIR Compiler 79

6.2.1 HorseCPU 81

6.2.2 HorseGPU 83

6.3 High-performance Built-in Function Library 85

6.3.1 Basic Built-in Functions 85

6.3.2 Important Database-related Functions 86

6.4 Data Management 92

7 Optimizations **93**

7.1 Early Optimizations 94

7.2 Type and Shape Analysis 96

7.2.1 A Motivating Example 96

7.2.2 Type Propagation . . . 99
7.2.3 Shape Propagation . . . 100
7.2.3.1 Shape Analysis . . . 103
7.2.3.2 Shape Propagation Rules . . . 105
7.2.3.3 Conformability Analysis . . . 109
7.3 Code Generation Optimizations . . . 113
7.3.1 Automatic Loop Fusion . . . 114
7.3.1.1 Fusion Nodes . . . 114
7.3.1.2 Code Generation for Vectors . . . 114
7.3.1.3 Generating Code for Lists . . . 115
7.3.1.4 Further Fusion Opportunities . . . 116
7.3.2 Fusing with Patterns . . . 116
7.4 Cross Optimizations . . . 121

8 Evaluations **123**
8.1 Experiment Setup . . . 124
8.2 Experiments with a Database Query Benchmark . . . 125
8.2.1 Benchmark Overview . . . 125
8.2.2 Complete Suite Results . . . 126
8.2.2.1 Results on sable-intel . . . 126
8.2.2.2 Results on sable-tigger . . . 134
8.2.2.3 Discussion . . . 137
8.2.3 Effect of Optimizations . . . 137
8.2.4 Scalability Study . . . 141
8.2.5 Compilation Time . . . 142
8.3 Experiments with an Array Language Benchmark . . . 144
8.3.1 Experiment Results . . . 145
8.4 Experiments with a UDF Benchmark . . . 147
8.4.1 TPC-H with UDFs . . . 148
8.4.2 UDF Derived from Black-Scholes . . . 149
8.5 Experiments with a GPU Benchmark . . . 155

8.5.1 Black-Scholes Results . . . 157
8.5.2 Morgan Results . . . 159
8.5.3 Discussion . . . 161

9 Related Work **163**
9.1 Database Query Processing . . . 163
9.1.1 Traditional Query Engines . . . 164
9.1.2 Modern Query Compilers . . . 164
9.2 Array Programming Languages . . . 166
9.3 Data Analytics in Database Systems . . . 168
9.4 Compiler Optimizations . . . 171

10 Conclusions and Future Work **173**
10.1 Conclusions . . . 173
10.2 Future Work . . . 174

A HorseIR Language Specification **177**
A.1 Language Grammar . . . 177
A.2 Value Ranges . . . 180
A.3 Built-in Functions . . . 182

B Plan-to-HorseIR Translator Specification **189**

Bibliography **195**

List of Figures

2.1 A primary key *deptid* in the table Department and a foreign key *empdeptid* in the table Employee 11
2.2 Example query derived from the TPC-H benchmark. 14
2.3 Example of execution plans for Figure 2.2 15
2.4 Support of an embedded UDF engine in database 16
2.5 Example query with a `scalar` UDF derived from Figure 2.2 17
2.6 Example query with a `table` UDF derived from Figure 2.2. 18
2.7 Examples of equivalent code in both array code and C code 20
2.8 Examples of unary and binary element-wise operations 21
2.9 Example of boolean selection 21
2.10 Example of array indexing (z=y(x)) 22

3.1 Overview of the HorsePower framework 26

4.1 Overview of HorseIR program structure 32
4.2 Example of a HorseIR module (bottom) for an SQL query (top) which returns the number of stores with relatively big discounts (50-80%). 33
4.3 Method name resolution when importing a module. 36
4.4 Example of list type 40
4.5 Example of the enumeration type: containing target, source, and index 41
4.6 Overview of the table type: containing a list of symbols and vectors . 42
4.7 Overview of the keyed table type: containing two normal tables . . . 43
4.8 Table conversion between normal and keyed tables 43
4.9 Specialization of a wild-card type with base and advanced types . . . 44

4.10 Example of a list function 45

5.1 Overview of HorsePower front-end which translates SQL queries, MATLAB functions, and MATLAB UDFs embedded in SQL code into HorseIR programs 51
5.2 Example of projection in HorseIR 53
5.3 Example of selection in HorseIR 54
5.4 Example of join in HorseIR 54
5.5 Illustration of the equal join operation in Figure 5.4 55
5.6 Example of an enumeration in a join with a pair of primary key (`deptid` in the table `Department`) and foreign key (`empdeptid` in the table `Employee`) 56
5.7 Example of enumeration in HorseIR 57
5.8 Illustration of the group operation in HorseIR 58
5.9 Example of an SQL query for group by (top) and its HorseIR code (bottom) 58
5.10 Illustration of the HorseIR code in Figure 5.9 59
5.11 Example of an SQL query for order by (top) and its HorseIR code (bottom) 60
5.12 Illustration of the order operation in HorseIR 60
5.13 Example of the join transformation: scenario 1 64
5.14 Example of an SQL query (top) and its HorseIR code (bottom) for the scenario 1 65
5.15 Example of the join transformation: scenario 2 65
5.16 Example of an SQL query (top) and its HorseIR code (bottom) for the scenario 2 66
5.17 Example of the join transformation: scenario 3 67
5.18 Example of an SQL query (top) and its HorseIR code (bottom) for the scenario 3 68
5.19 Generating HorseIR code from MATLAB within the McLab framework 68
5.20 Example of MATLAB 72

5.21 Example of the MATLAB function with vectors as input 72
5.22 HorseIR code with a new method for the UDF in Figure 2.5 75

6.1 Overview of HorseIR back-ends 77
6.2 Overview of HorseCompiler 80
6.3 Overview of HorseIR working with GPUs 83
6.4 Example of OpenACC code 84
6.5 Example of the compress function 86
6.6 Design of array-lookup join 87
6.7 Design of radix-based hash join 88
6.8 Global table registration and fetching 92

7.1 Analysis and code generation overview. 94
7.2 Example query and its HorseIR program 97
7.3 Optimized C code for the IR code in Figure 7.2b 98
7.4 Type nodes for representing type information 99
7.5 Type rules for boolean binary functions with two parameters x and y 101
7.6 Generated fused C code 103
7.7 Example propagating the scan shape 107
7.8 A fusible section for the HorseIR program in Figure 7.2b. The text format on the right hand side is `<statement>(<group>): <variable>::<shape>`. 113
7.9 Code generation for vectors. `Rop`: reduction operation; `Rfinal`: final reduction step (e.g. divide by element count); `z`: accumulator/output vector. 114
7.10 Code generation for lists. `Rop`: reduction operation; `Rfinal`: final reduction step (e.g. divide by element count); `t`: cell accumulator; `z`: output vector. 115
7.11 Code examples for fusing with patterns 116
7.12 Illustration for generating fused code with FP-1 117
7.13 Illustration for generating fused code for FP-2 118

7.14 Illustration for generating fused code with FP-3 118
7.15 Illustration for generating fused code with FP-4 119
7.16 Patterns designed for FP-2 and FP-3 120
7.17 Dependence graphs for the example in Figure 5.22 to show that method inlining helps explore more opportunities for automatic loop fusion. . 122
7.18 Generated C code after the cross-method optimization in Figure 7.17 122

8.1 Overview of experiments used to evaluate the performance of HorsePower 123
8.2 TPC-H schema and its table sizes (row, column) on SF1. 126
8.3 (sable-intel) Performance comparison between MonetDB and HorsePower over all TPC-H queries with 1 GB input data (SF1) 128
8.4 (sable-intel) Results of individual queries and threads in terms of geometric mean. 133
8.5 (sable-tigger) Performance comparison between MonetDB and HorsePower over all TPC-H queries with 1 GB input data (SF1) 135
8.6 (sable-tigger) Results of individual queries and threads in terms of geometric mean. 136
8.7 Geometric mean execution time for HorseIR and MonetDB across five different SFs on sable-intel. 141
8.8 Compilation time on both sable-intel and sable-tigger for TPC-H queries 142
8.9 Base queries with scalar and table UDFs 152
8.10 Example queries of variation 1 with scalar and table UDFs 153
8.11 Example queries of variation 2 with scalar and table UDFs 154
8.12 Example queries of variation 3 with scalar and table UDFs 155
8.13 Performance breakdown in Black-Scholes for both the naive and optimized versions. 158
8.14 Performance breakdown in Morgan for both the naive and optimized versions. 160

List of Tables

4.1 List of HorseIR base types 37
4.2 HorseIR advanced types 39

5.1 Type mapping from MATLAB to HorseIR 71

7.1 Definitions of vector shapes 103
7.2 Rules for binary element-wise Functions (E) 106
7.3 Rules for unary element-wise functions (E) 106
7.4 Rules for reduction functions (R) 106
7.5 Rules for the scan function (S) 107
7.6 Rules for array indexing (X) 108
7.7 Rules for special boolean functions (B) 108
7.8 Conforming rules for two shapes 110

8.1 Overview of machines used in experiments 124
8.2 Profiling data of all TPC-H queries: tables and the number of joins, predicates, aggregations, groupbys, sorts, and return columns 127
8.3 (sable-intel) Performance speedups on SF1 over No-opt obtained by various HorseIR optimizations for different queries. 138
8.4 (sable-tigger) Performance speedups on SF1 over No-opt obtained by various HorseIR optimizations for different queries. 140
8.5 Compilation time breakdown in seconds and percentage 143
8.6 Speedup of HorsePower over MATLAB in execution time using Black-Scholes (in milliseconds) 146

8.7 Speedup of HorsePower over MATLAB in execution time using Morgan (in milliseconds) 146
8.8 Execution times and speedup (SP) of HorsePower over MonetDB using the modified TPC-H benchmark with UDFs. 148
8.9 Black-Scholes execution time Python vs. HorseIR 149
8.10 Execution time and speedup (SP) of HorsePower (HP) compared to MonetDB (MDB) for variations in Black-Scholes. 151
8.11 Overview of the portions for VersionCPU and VersionGPU 156
8.12 Black-Scholes: Performance comparison between VersionCPU and VersionGPU for both the naive and optimized versions over multiple threads (1 to 16). 157
8.13 Morgan: Performance comparison between VersionCPU and VersionGPU for both the naive and optimized versions over multiple threads (1 to 16). 159

List of Abbreviations

AST	abstract syntax tree
DB	database system
IR	intermediate representation
IMDB	in-memory database system
JSON	JavaScript object notation
RDBMS	relational database management system
SQL	structured query language
UDF	user-defined functions

Chapter 1
Introduction

Two recent developments in the last decades have considerably shifted the focus of query optimization for relational database management systems (RDBMS). First, main memory has become consistently cheaper making it possible to have even large databases reside in memory. Thus, research has shifted their focus on main-memory- and CPU-based query optimization where the disk is no longer considered a bottleneck. Furthermore, the reorganization of the storage structure, where tables are no longer stored on a per-row basis but instead each column is stored in an array-like structure, has shown to have tremendous performance benefits for analytical workloads. As such, query execution has become much more like the execution of standard programs, in particular, of programs written in array-based programming languages. Given the rich set of compiler optimization techniques that have been developed for such high-level languages, we see a great opportunity to exploit some of them for query processing.

The database research community has started to look at compiler solutions in the recent past. The "array like" nature of a table's column in column-based RDBMSes, especially its working data sets[1], naturally led to a series of studies and optimization strategies focused on the benefits of CPU caching [16, 46]. However, to the best of our knowledge, beyond these excursions into exploiting CPU cache-based optimizations,

[1]The term working dataset is used to denote the copy of data that has been brought from the disk to main memory for processing a request.

the database community at large is yet to benefit from more comprehensive optimization techniques that the compiler community has amassed from its decades of research on array programming languages. We believe that working datasets of column stores are good candidates to apply array programming optimizations, as a column essentially contains homogeneous data which maps nicely to array-based/vector-based primitive functions.

In addition, in the era of Big Data, complex data analytics that goes well beyond SQL-only declarative queries, has become increasingly important. Although the amount of data stored in traditional RDBMSes has been growing rapidly, the by far most common current approach to perform such analytics is to take the data first out of the database system and load it into stand-alone analytical tools, which are often integrated programming language systems, such as Python, MATLAB [56], and R [10]; or specialized scalable analytical platforms such as Map/Reduce [27] and Spark [92]. Many of these programming languages support array programming, such as Python/NumPy [6] and MATLAB. When needed, the results of such a database system external analysis can be reintegrated into the database. However, as the size of the data increases, the expensive data movement between database systems and data analytics tools can become a severe bottleneck.

An alternative, that avoids such data movement, is to integrate the analytical capabilities into the database system. The most well-known approach to do so is to support user-defined functions (UDFs) written in a conventional, high-level programming language, that are then embedded in SQL queries [72, 59, 90, 83]. For example, MonetDB [72] allows users to include Python functions into their SQL queries. These functions are then executed by a language interpreter (Python) that is embedded inside the database system engine.

Although UDF implementations connect SQL queries and high-level programming language functions, they still have separate execution and optimization environments: one being the SQL execution engine, and the other the programming language execution environment. This can quickly lead to costly data format conversion between the two environments. As such, we believe it is necessary to have an approach where both the declarative query component and the analytical functions are optimized and

executed in a holistic manner.

1.1 Objectives

The goal of this thesis is to develop an approach that allows us to exploit the rich set of compiler optimizations that have been developed for array-based languages, to be used for database queries and data analytics. Since database queries as well as the underlying execution plans that implement these queries in current RDBMS look very different than array-based programs, this thesis proposes an intermediate array-based representation that is suitable to adequately represent query execution plans. Its design also considers the high-level source languages that can support user-defined functions. Therefore, it is also an intermediate representation that can be used for various source languages in the context of programming languages. The focus of this thesis lies in the design and implementation of this language and its compiler. By employing a set of compiler optimizations for array-style code, we aim in generating efficient target code for multiple platforms.

1.2 Approach Overview

In this thesis, we present a framework called *HorsePower* for the execution of database queries and data analytics functions. In order to support database queries, we design and implement a new array-based intermediate representation (IR), called *HorseIR*, which is the core part of HorsePower. The idea of generating array-based code from database queries is both novel and challenging. To the best of our knowledge, we are the first to propose such a relatively high-level array-based IR that represents database query processing and allows column-based in-memory database systems to benefit from a whole range of compiler optimizations.

The core data structure in HorseIR is a vector (corresponding to columns in the database tables) and the implementation has a rich set of well-defined high-level built-in functions with clearly defined semantics. They are easy to optimize, and in the

case of element-wise built-in functions, are easy to vectorize and parallelize. The type system of HorseIR facilitates declaring variables with explicit types, as well as a wild-card type and associated type inference rules. Important relational operations, such as projection, selection, join, and aggregation, can be represented with one or more HorseIR built-in functions. Rather than implementing complex operations such as database join as one big function, HorseIR provides a repertoire of smaller built-in functions that can be combined to achieve the same functionality. For example, HorseIR has a function `join_index` which returns the indices of the joined columns that can be further utilized by subsequent operations.

We believe that introducing a high-level IR might open a new research direction for database query optimizations. However, introducing an IR may also result in some overhead and must be considered along with the performance benefits that they bring.

Other than database queries, HorseIR can represent MATLAB programs that work on one-dimensional arrays. Some of the built-in functions designed for database queries, such as `sum` for computing the total value, can be used for MATLAB programs as well. Furthermore, we have added some specialized functions common for analytics, such as `cumsum` for computing cumulative sum. Thus, user-defined functions (UDFs) written in MATLAB can be translated to HorseIR. Additionally, when they are embedded in SQL queries, both the SQL and the MATLAB components are translated together into a single HorseIR program facilitating holistic optimizations.

HorsePower has a set of facilities for generating and optimizing HorseIR code. We provide automatic translators for generating HorseIR from SQL and MATLAB. For SQL, we follow a layered approach that facilitates a wide range of compiler optimizations in a systematic way. It exploits and further builds upon the many optimizations developed by the database community in terms of generating efficient execution plans for declarative queries. Specifically, we propose an approach where SQL queries are first translated into execution plans using standard database optimization techniques that consider the operators in the query and the characteristics of the input dataset. These database optimized plans are then translated to HorseIR. After generating HorseIR code, proper compiler optimizations, such as fusion-based optimizations, can be

applied to HorseIR, and its code can be thereafter compiled to efficient target code for multiple platforms, such as CPUs and GPUs, before generating executable files. In addition, HorsePower has a data management component to support the execution of the generated binary code on datasets stored in files.

1.3 Contributions

We summarize the contributions of our thesis as follows:

Design and implementation of HorseIR. We first identify the advantage of an array-based IR for database query processing. We propose *HorseIR*, an array-based IR, and implement the IR to represent both database queries and array programming languages.

Automatic translator support for generating HorseIR. We deliver translators which can generate HorseIR code from database execution plans automatically. This work is novel as we are the first bridging the gap between database query and array-based languages, despite the large syntax difference between the two languages. Moreover, we implement a translator for compiling MATLAB, an array language, to HorseIR. The translator connects two array-based languages, though they are designed for different domains.

UDF support with HorseIR. We support UDFs embedded in database queries by compiling both the UDF and the SQL component to HorseIR code and then integrate the two code snippets into one HorseIR program. Thus, the challenging problem of optimizing database queries with UDFs is handled by creating a single HorseIR program that can be optimized holistically.

Tailored optimizations. We identify and apply a set of compiler optimizations for generating efficient target code from HorseIR. Thus, we improve database query performance by applying techniques derived from array programming. We perform a precise shape analysis on the built-in functions of HorseIR that

represent database operators. This allows us to collect accurate shape information for subsequent loop-fusion based optimizations. Furthermore, we prepare a set of pre-defined code patterns for generating efficient code.

Multiple-platform support. We build different back-ends in HorsePower with the purpose of supporting target code generation for various parallel hardware, i.e., target C code that runs well on CPUs and GPUs, respectively. By having a single IR we have the flexibility to cope with the rapid development of hardware platforms.

Performance evaluation. We conduct experiments on standard database benchmarks, statistical benchmarks, and query UDF benchmarks to show the effectiveness of our techniques. We demonstrate performance benefits of compiler optimizations, and compare the overall performance of our system with the popular column store based RDBMS, MonetDB.

1.4 Publications

Content presented in this thesis has been previously published in the papers listed below. There is no exact match to chapters as the thesis has been designed in a holistic manner and the contributions of any of these papers is spread across several chapters.

1. Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren, ***HorseIR: Bringing Array Programming Languages together with Database Query Processing***, Proceedings of the 14th Symposium on Dynamic Languages (DLS'18), pp. 37-49, November 2018.

This paper is the foundation of this thesis and introduces the design and implementation of our array-based intermediate representation, *HorseIR*, for database SQL

queries. We provide a translator to generate HorseIR from execution plans generated for SQL queries, and a compiler that optimizes HorseIR and generates efficient code using some initial optimization mechanisms. As a main contributor to the paper, I mainly worked on the design and implementation of HorseIR, the source-to-source translator for generating HorseIR from execution plans, and the compiler for emitting efficient target code. Experiments were conducted by me. Joseph Vinish D'silva and Bettina Kemme joined the research with their expertise in database domains, and provided important suggestions and feedback for experiments; Hongji Chen helped with the initial design of HorseIR; and Laurie J. Hendren supervised this research, joined discussions, and provided valuable feedback.

2. Hanfeng Chen, Alexander Krolik, Bettina Kemme, Clark Verbrugge, and Laurie Hendren, ***Improving Database Query Performance with Automatic Fusion***, Proceedings of the 29th International Conference on Compiler Construction (CC'20), pp. 63-73, February 2020. (**Best paper finalist**)

This paper presents a compiler approach to optimize execution plans that are expressed in HorseIR. By analyzing the shape properties of programs, we employ a novel data flow analysis in order to collect information for the subsequent optimization phase, which fuses multiple operations to generate code with less loops, reducing the need for separate computation and the storage of intermediate values. As the main contributor of this paper, I designed and implemented this optimization strategy and performed the experiments. I discussed with Alexander Krolik and received his valuable feedback. He also helped clarify the research ideas and write the paper. Bettina Kemme and Clark Verbrugge joined the discussion of the paper, provided feedback, and helped with the paper writing. Laurie J. Hendren supervised this paper and shared her insights in the optimization strategy in the early discussion that played a vital role in the paper.

3. Hanfeng Chen, Joseph Vinish D'silva, Laurie Hendren, Bettina Kemme, ***HorsePower: Accelerating Database Queries for Advanced Data Analytics***, Proceedings of the 24th International Conference on Extending Database Technology, (EDBT'21), pp. 361-366, March 2021.

This paper proposes the extension of HorseIR to support database queries with user-defined functions (UDFs) for data analytics. We first provide a translator which translates MATLAB programs to HorseIR code. Furthermore, we extend our SQL to HorseIR translation process to allow for SQL queries with embedded MATLAB UDFs, and perform holistic optimizations. As the main contributor, I worked on the design and implementation of the system, and conducted experiments. Joseph Vinish D'silva shared his insightful feedback, gave suggestions on the system design and implementation, and helped with writing the paper; Laurie J. Hendren joined at the early stage of the research and provided valuable feedback; and Bettina Kemme contributed her time to supervising the whole process of the research, including research discussions and paper writing.

1.5 Thesis Organization

The rest of the thesis is organized as follows. We first introduce the background of our research in Chapter 2. Next, we provide an overview of the HorsePower system in Chapter 3. In the following chapters, we present the details of the system including: the design and implementation of HorseIR in Chapter 4; the system front-end for generating HorseIR from various source languages in Chapter 5; the system back-end for generating multiple target code from HorseIR in Chapter 6; and compiler optimizations in Chapter 7. After that, we show the evaluation of our experiments in Chapter 8, and related work in Chapter 9. Finally, we conclude and discuss possible future work in Chapter 10.

Chapter 2
Background

In order to bridge database query processing and array programming languages, we need to understand a wide range of concepts from both research fields. We first introduce SQL query processing and user-defined functions in relational database systems in Section 2.1. Next, we present array programming languages in Section 2.2 in order to better understand the design behind HorseIR. Finally, we explain the concept of intermediate representations in Section 2.3.

2.1 Query Processing in Relational Database Systems

Relational Database Management Systems (RDBMS) have been the primary data management software of choice for organizations for decades. In this section we first introduce the relational model used in RDBMS in Section 2.1.1; database storage formats in Section 2.1.2; SQL, the de facto standard query language, and relational algebra, on which SQL is based on, in Section 2.1.3; and the steps of processing an SQL query in Section 2.1.4. Finally, we present the concept of user-defined functions and how they can be embedded in SQL queries in order to enable data analytics within a RDBMS in Section 2.1.5.

2.1.1 Relational Model

The relation model was first introduced by Edgar Codd [25]. It defines the organization of data as *relations*, consisting of a set of tuples. A relation can be viewed as a database table whose rows are the tuples and whose columns are the attribute values. Using a vertical view, a database table contains a list of columns, each of which has homogeneous data.

A relational database is a database whose data is organized following the relational model. It may contain multiple tables that are connected via primary and foreign key relationships. A primary key of a table consists of one or more attributes, such that each tuple of the relation has different values in those attributes. That is, primary key values are unique. A foreign key of a table is a set of attributes that refer to a primary key in a different relation. Note that foreign keys are not required to be unique. For instance, a relational database may have the following tables:

- **Department** (deptid, deptname)
- **Employee** (empid, *empdeptid*, empname)

Two tables are created for storing department and employee information. The table `Department` has two columns: department ID (`deptid`) and name (`deptname`), in which `deptid` is a primary key. Table `Employee` has a column with the employee ID (`empid`) as its primary key, and two other columns indicating a department (`empdeptid`) and a name (`empname`). The column `empdeptid` is a foreign key referring to a specific department (i.e., its value is one of the values of the primary key `deptid` of the department table). It refers to the department in which the employee works.

As can be seen in Figure 2.1, the employee "Sam" is associated with department ID 76 which refers to the department "DeptJava". Moreover, the employees "Kate", "John", and "Fang" are from the same department "DeptC" because they share the same department ID 14.

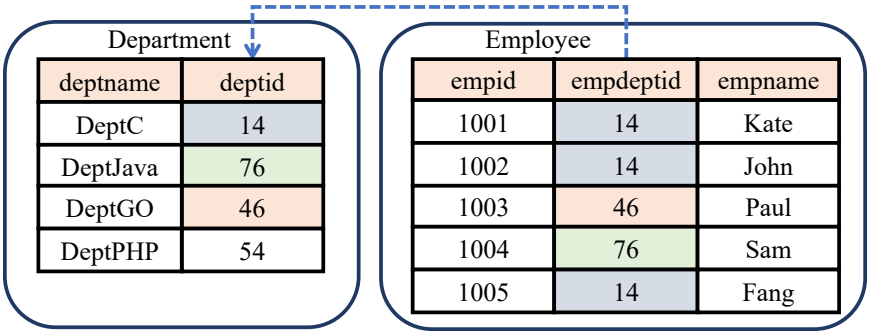


Figure 2.1 – A primary key *deptid* in the table Department and a foreign key *empdeptid* in the table Employee

2.1.2 Database Storage

Most existing RDBMSes store tables in row format where all attributes of a row are stored in contiguous space. In contract, more recent systems follow a columnar storage format, where each column of the table is stored in contiguous space. Row-based systems are better when queries access most of the attributes of the table and also when there are many inserts and updates. Columnar systems have shown superior performance for analytic workloads on large tables, where the data is seldom or not at all updated, such as the historical data from the financial market. It should be noted that a database system may support both formats in order to leverage the advantage of these formats for applications under different scenarios, such as the commercial database system SAP HANA [32].

2.1.3 Relational Algebra and SQL

In order to query and manage relational data, RDBMS support SQL (structured query language) which is a domain-specific language based on relational algebra. SQL is a very high-level language that provides a declarative way to write database queries. The query describes what data should be retrieved without the need to know the underlying database implementation.

RDBMS usually parse an SQL query into a relational algebraic representation [17], as the latter has been known to be easier to optimize [79]. Relational algebra contains

a set of important operations, such as *projection* for loading columns from tables, *selection* for filtering qualified rows with conditions, *aggregation* for grouping rows with the same value, and *join* for merging two tables over columns with conditions. Such relational algebra operators are unary or binary, in the sense, they accept one or two tables as input. Their output is always a single table. In here we shortly introduce each of them and show the SQL equivalent.

Projection. A projection operation is defined as $\Pi_{c_1,c_2,...,c_n}(R)$ which takes the records of table R as input and returns the same records but only the columns of R with column names $c_1, c_2, ..., c_n$. Projection refers to the *SELECT* clause of an SQL query retrieving columns from tables, e.g., `SELECT c1,c2,...,cn FROM R`. This returns a table that contains columns `c1,c2,...,cn`. If all columns need to be projected, then one uses `SELECT * FROM R;`.

Selection. A selection operation is denoted as $\sigma_P(R)$ where P is a collection of selection predicates and R is a table. The selection returns those records of R whose attribute values fulfill the condition P. Formally, $P = (P_1 < op_1 > ... < op_n > P_n)$ where op_i is either $\wedge$ for a logical `AND` operation or $\vee$ for a logical `OR` operation, and $\{P_1, ..., P_n\}$ are predicates on attributes. P is represented in the WHERE clause of an SQL query e.g., `SELECT * FROM R WHERE c1 < 100 AND c2 = 10;`.

Join. A join operation takes two tables as input and connects records from the two tables that fulfill certain conditions. Let R_1 be a table with columns $(c_{a_1}, ..., c_{a_n})$, and R_2 be another table with columns $(c_{b_1}, ..., c_{b_m})$. Then, the join operation returns a new table:

$$R_1 \bowtie_{COND} R_2$$

where $(COND \leftarrow (c_a \ cond\_op \ c_b) \wedge \ ...)$ and `cond_op` can, for instance, be any of the operators for comparison, such as equal ($=$), not equal ($\neq$), less than ($<$), or greater than ($>$). The new table contains the columns from both the tables R_1 and R_2. A record r_1 from R_1 together with a record r_2 from R_2 build a record in the new table, if r_1 and r_2 fulfill the conditions defined in $COND$. As

an example, assume $COND \leftarrow (c_{a1} = c_{b2})$, this is expressed in SQL as `SELECT * FROM` R_1`,`R_2 `WHERE` c_{a1}`=`c_{b2}`;`, and the join is called an *equijoin* as all (one) conditions are equality comparisons.

Aggregation. An aggregation function takes a list of values as input and returns a single value as output. Examples are `sum` (sum of values) and `count` (number of values). A formal definition of aggregation is

$$AGGR_{F_1(c_1),F_2(c_2),...,F_m(c_m)}(R)$$

where (i) $c_1, c_2, ..., c_m$ are names of columns in R; and (ii) $F_1, F_2, ..., F_m$ are aggregation functions. For instance, $AGGR_{SUM(c_1)}R$ is expressed as `SELECT SUM(c1) FROM R;` in SQL.

Group by. Using a *group by* (referred to as `groupby` in this thesis) $G_{c_1,c_2,...,c_n}(R)$, all tuples that have the same values in columns $c_1, c_2, ..., c_n$ are grouped and will create one output tuple. The groupby can only be used in combination with aggregation functions and projections on the attributes in the groupby clause, for instance, $\Pi_{c_1,sum(c_2)}(G_{c_1}(R))$. In SQL notation, this is written as `SELECT c1, SUM(c2) FROM R GROUP BY c1;`. The output is one tuple for each set of tuples in R that have the same value in `c1` and this value is returned together with the sum of (probably different values) this set of tuples has in column `c2`.

Order by. An *order by* sorts all input tuples based on one or more particular columns since a relation has no pre-defined ordering. A typical example in SQL is `SELECT c1,c2 FROM R ORDER BY c1 ASC;` where the rows are sorted in ascending order of column `c1` and then `c1` and `c2` are returned. The sorting order for one column can be either ascending or descending.

An SQL query can contain many selections, joins, aggregations, and projections. We will see throughout the thesis more complex examples and we will discuss in detail as we present them.

```
1 SELECT
2   SUM(l_extendedprice * l_discount) AS RevenueChange
3 FROM
4   lineitem
5 WHERE
6   l_discount >= 0.05;
```

Figure 2.2 – Example query derived from the TPC-H benchmark.

2.1.4 Database Query Processing

As the input of relational algebra operators are one or two tables, and the output is always a table, it is easy to chain these operators into operator trees, also called *execution plans*, where the output of one relational operator serves as the input of another operator in order to solve complex SQL queries. Each of the operators can be implemented in various ways. Modern RDBMS optimizers have a query re-write subsystem that generates multiple semantically equivalent execution plans [42, 43] for a given query. They might differ in the order of the operators and the implementations of the individual operators. The performance of execution plans can vary widely and depend on the input data. The overall cheapest execution plan is then chosen based on cost models. In this aspect, one can think of an RDBMS as a dynamic language processor which receives SQL queries as input and dynamically translates the SQL query to an optimized execution plan that minimizes the execution cost of the query and executes the plan.

Figure 2.2 presents a typical SQL query derived from the TPC-H benchmark [85]. It works on a table `lineitem` to get the total revenue change when discounts are greater or equal than 0.05. Figure 2.3 shows the execution plan which contains several operators, including a projection on the relevant columns `l_discount` and `l_extendedprice`, a selection for finding qualified rows in which discounts are greater or equal than 0.05, an aggregation operation `SUM` to compute the total revenue change, and a final projection to retrieve the result in a column named `RevenueChange`.

An SQL query can be processed using two different execution models, the *Volcano iterator* model [37] or the *data-centric* model [64]. The Volcano model represents

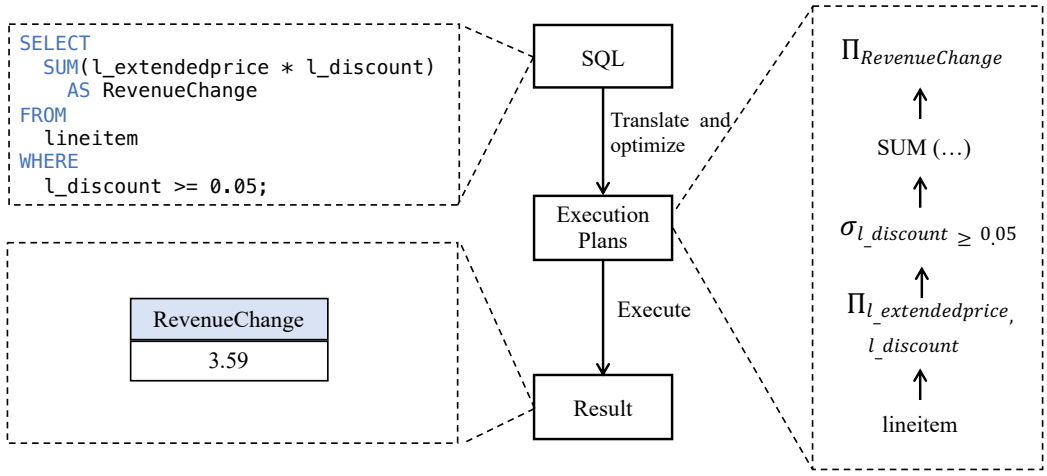


Figure 2.3 – Example of execution plans for Figure 2.2

the classical approach which fetches tuples through a set of chained operations in a pipelined fashion. This avoids large intermediate results as the tuples are produced as needed. The Volcano iterator model was developed with a row-based storage in mind. In contrast, the data-centric model has gained recent attention with the development of modern query compilers. This approach allows data to be pushed through an operator completely. It is popular with columns storage systems, where an operator, e.g., a selection, is executed on all elements of a column. To avoid intermediate results, it is important to have an optimizer that is able to determine groups of operators that can be grouped and merged to achieve better data locality and reduce intermediate results. Although a direct comparison is absent, such data-centric approaches promise to deliver better performance [76]. The design of HorseIR is akin to the data-centric model where arrays of data are passed from one function to the next, and its optimizer employs a fusion-based strategy that merges functions to generate compact target code.

2.1.5 User-defined Functions in Database Queries

A user-defined function (UDF) is a high-level language function embedded within an SQL statement. It can simplify the query by offloading some of the computation in a

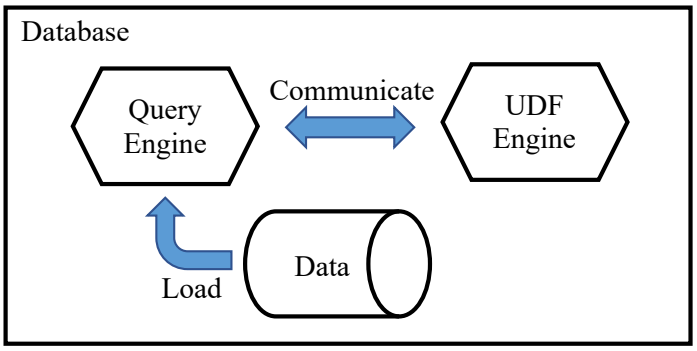


Figure 2.4 – Support of an embedded UDF engine in database

more concise language other than SQL, or it can provide additional functionality that cannot be expressed by SQL alone. Some RDBMSes offer their own vendor-specific SQL language extensions to write UDFs, such as Transact-SQL UDFs in Microsoft SQL Server [73]. Moreover, many database systems provide interfaces for integrating general-purpose programming languages into SQL queries, such as the embedded Python interpreter in MonetDB [72]. As depicted in Figure 2.4, the embedded UDF engine needs to communicate with the query engine of the database system in order to get input data and write out the result. The cost of communication could be expensive due to the data conversion between two different engines. For example, Python can only operate on Python objects, requiring columns or rows to be counted to Python objects. Nevertheless, using well-known programming languages for UDFs has become a popular choice as it simplifies software development.

UDFs are often classified into subcategories depending on their expected interaction with the SQL query. For the sake of brevity, we will focus only on *Scalar UDFs* and *Table UDFs*, as these are the most commonly employed types of UDFs and also the ones supported presently in HorsePower. A detailed discussion on the various UDF categories can be found in [72].

Scalar UDFs A *scalar UDF* returns a single value (which could be a vector) and can be therefore essentially used wherever a regular table column is used, such as in the `SELECT` or the `WHERE` clause of SQL queries. Figure 2.5 shows a scalar UDF

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

```
1 SELECT
2     SUM(calcRevenueChangeScalar(l_extendedprice,l_discount)) AS
    RevenueChange
3 FROM
4     lineitem
5 WHERE
6     l_discount >= 0.05;
```

Figure 2.5 – Example query with a `scalar` UDF derived from Figure 2.2

which performs the multiplication that was originally part of the `SELECT` clause in Figure 2.2. Although this is a simple example, outsourcing this computation to a UDF extends its use across several queries, and allows for a simpler change of the implementation or semantics of the UDF (and therefore, that of the queries using it).

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

In contrast, in a column-based system, and under the assumption that the programming language used for the UDF can handle array-based data structures, the execution is quite different. Let's look at the execution within MonetDB. First, the `WHERE` clause is executed on the entire `l_discount` column, returning a boolean vector of the same size as `l_discount` with true values in the elements (rows) that fulfill the condition. Then, MonetDB applies the corresponding boolean selection on columns `l_discount` and `l_extendedprice` resulting in "compressed" columns only containing the elements of `l_discount` and `l_extendedprice` for which the corresponding entry in the boolean vector was true. These two compressed columns are then given to the UDF as arrays, and the UDF performs an element-wise multiplication on these arrays returning an array of the same size. This is then the input to the `SUM` operator. Thus,

compared to a row-based system, the UDF is only called a single time and works on arrays instead of individual values.

```
FUNCTION calcRevenueChangeTable(price,discount)
    mask = discount >= 0.05;
    revenuechange = price[mask] * discount[mask];
    RETURN TABLE("revenuechange", revenuechange);
END
```

```
1 SELECT
2     SUM(revenuechange) AS RevenueChange
3 FROM
4     calcRevenueChangeTable((SELECT l_extendedprice, l_discount FROM
     lineitem));
```

Figure 2.6 – Example query with a `table` UDF derived from Figure 2.2.

Table UDF A *table UDF* returns a table-like data structure, and thus, is typically called within the `FROM` clause of an SQL statement, similar to regular database tables. As such, it can return one or more columns at the end of its execution. Figure 2.6 shows a table UDF which is specifically designed for a column-based system. The input for the table UDFs is the `l_extendedprice` and `l_discount` columns of the `lineitem` table. Similar to the scalar UDF example, the database performs the selection operation based on the value of the `l_discount` column. It then passes the values of the `l_extendedprice` and `l_discount` columns for the selected rows to the table UDF. As such, each input column to the table UDF is a vector of values for the corresponding columns, equal in length to the number of selected records from the `lineitem` table. The table UDF then performs the element-wise multiplication function, and returns a table-like data structure with the result of this multiplication as the column `revenuechange`. This resulting table-like data structure is then the input of the surrounding SQL query, which uses it in its `FROM` clause and executes the `SUM` operation on `revenuechange`.

Performance The main advantage of using UDFs is that the core computation of the query is abstracted into a function. Whether to use a scalar or a table UDF

depends on the expertise of the developer and task to be performed. In the case of column-based systems and the use of programming languages that support operators on arrays, such as MATLAB, R, or Python, column-based data can be exchanged seamlessly between SQL and the embedded UDF, and efficient array operations can be exploited within the UDF.

However, introducing UDFs into queries can result in performance issues. Firstly, the overhead of possible data movement and conversion is non-negligible when data materialization is required for the input of a UDF and as well when the return value of a UDF is given back to the SQL statement. This is because often the data types used by the two execution environments are not similar, requiring the implementation to perform data conversion from one format to the other. In fact, MonetDB tries to avoid this by employing a concept called *zero-copy* [54], whereby a high-level language UDF can directly access the database system buffers of a column if the underlying binary structures of the data types used in both the UDF and the database system are compatible.

Apart from the data format issue, the fact that there is a large syntax gap between SQL and the UDF language results in both parts of the query being executed as black boxes to each other by two separate execution environments. In other words, there is typically no cross optimization between the two components of the query. An exception is the approach in Froid [73] having UDF code to be rewritten to SQL code so that the query optimizer of the database system can optimize across the entire rewritten query. This is possible with the examples that we discussed above, as the tasks performed by the UDFs can be expressed as SQL operators. However, this is not the case for all the tasks for which UDFs are currently used, such as those involving non-relational operations on the data.

2.2 Array Programming Languages

Array programming is supported by a wide range of programming languages, such as MATLAB, APL, and FORTRAN 90. The main characteristics of array programming

are: (i) Array objects are the main data structure. An array object is able to represent an arbitrary dimensional array. As a consequence, programming with arrays comes with succinct and expressive code; and (ii) Array programming languages provide a rich family of well-defined operators as built-in functions.

ID	Array Code	C Code

(1)	`z = ( x + 0.5 ) * y;`	`for(i=0; i<size; i++){`
		`  z[i] = (x[i] + 0.5) * y[i];`
		`}`

(2)	`z = y(x);`	`for(i=0; i<size_x; i++){`
		`  z[i] = y[x[i]];`
		`}`

Figure 2.7 – Examples of equivalent code in both array code and C code

Figure 2.7 presents equivalent code snippets in array and C code for (1) basic arithmetic operations on equal-length vectors; and (2) array indexing on the two vectors returning a new vector with the same length as the vector `x`[1]. As can be seen, the array code is more expressive than C code, and it is easy to generate C code from the array code. The fundamental idea of array programming is to apply an operation on all items of an array without an explicit loop. There are two important operations in array languages: element-wise operation and boolean selection.

Element-wise operations apply a function to data stored in input arrays in an element-wise manner. An element-wise operation may take one (unary) or two (binary) arguments depending on the function it implements. Examples are shown in Figure 2.8. The unary operation `abs` processes each value in the array and generates a new array as output containing corresponding absolute values; the binary operation `plus` has more cases distinguishing whether both input arrays agree on the length or one of them has a single element. Element-wise operations can be used for query processing in column-store RDBMS when database operations should be applied to a whole column. In terms of performance, they can be executed in parallel as they are data dependency free. For instance, MATLAB's element-wise built-in functions are well-tuned for implicit data parallelism. We will provide more details when we

[1] Array bounds checking will be required in actual C code.

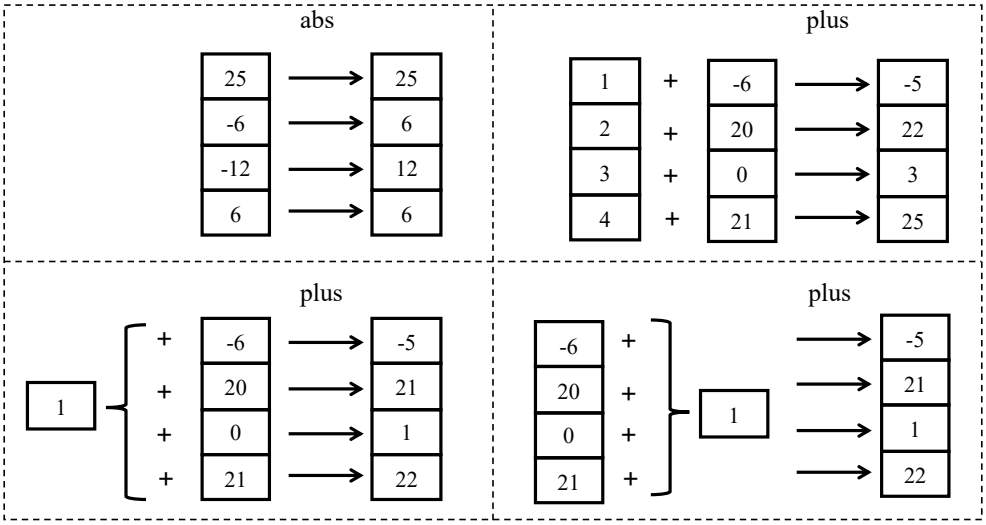


Figure 2.8 – Examples of unary and binary element-wise operations

introduce HorseIR's element-wise built-in functions in Section 4.4.

A special concept in array programming is vectorization. It can take place in the low-level hardware as well as the high-level programming language. Modern hardware is actively adopting the concept of vectorization in their chip design. For instance, Intel Advanced Vector Extensions (AVX) [41] is a hardware-based instruction set designed for efficient vector operations. One instruction performs one operation on multiple data items simultaneously. A software-based version of vectorization is the source-level translation from a scalar form to a vectorized form to reduce the overhead of explicit loop iterations [58, 21]. With vectorization, the performance of element-wise functions can be further improved.

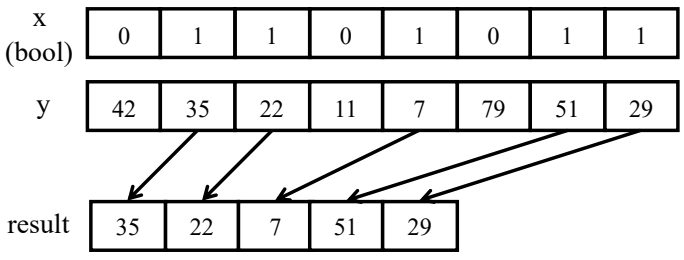


Figure 2.9 – Example of boolean selection

Boolean selection is another important operation in array programming. It takes two parameters: a boolean array (often referred to as "mask") and another equal-length array, and returns the elements of this second array where the boolean vector has in the same position a true value. As can be seen in Figure 2.9, the result is determined by the "true" value (i.e., 1) in the boolean array, and the size of the result is equal to the number of true elements in the boolean array. We will see that we can use boolean selection for SQL queries when we perform a projection after a selection. Different array programming languages may have different notations or names for this operation. Our HorseIR provides a built-in function called *compress* for the boolean selection.

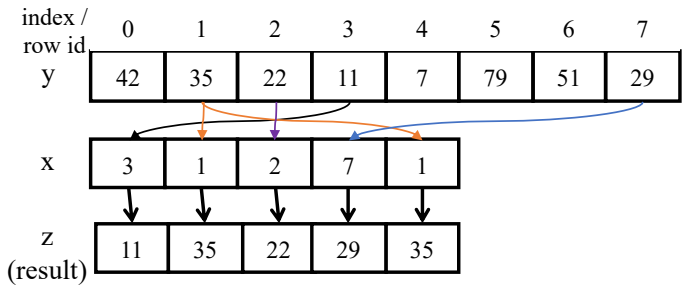


Figure 2.10 – Example of array indexing (z=y(x))

Array indexing is commonly used in array programming. As can be seen in Figure 2.10, given an array x consists of indices where $x = \{x_0, x_1, \ldots, x_n\}$, a new array z is generated on the input array y as $z = \{y[x_0], y[x_1], \ldots, y[x_n]\}$. An index indicates the "position" of the value which needs to be fetched. The size of the resulting array is determined by the size of the indexing array. In database systems, such indices are conceptually same to "row ids" as well. The row ids are implicitly existed and they are critical for RDBMS to fetch new values from row ids. In particular, a column-based RDBMS often first scans some columns in tables to find rows which fullfil given conditions, and then returns new tables with all columns but only preserving these qualified rows using row ids.

Built-in functions in array programming languages are also called *primitive functions*, and represent a set of basic computations. It is common to have new built-in

functions for encapsulating new functionality. In fact, the number of built-in functions in array languages can vary greatly with specific needs in their domains. One performance problem for built-in functions is the intermediate results after each function invocation. For instance, the first array code in Figure 2.7 needs to be executed in two steps: (i) `t = x + 0.5` and (ii) `z = t * y`, where the variable `t` is a temporary vector. Thus, the performance of the array code is slower than that of the C code because the C code has no intermediate vector. To improve the performance of array programming with built-in functions, it is common to have fusion-based optimizations, such as loop fusion, to reduce the number of loops generated for these functions and eliminate intermediate results [47, 89].

2.3 Intermediate Representations

In compiler design, it is common to introduce an intermediate representation (IR) as an intermediate layer between a source language and generated target code [61]. Having a common IR can reduce the complexity of supporting multiple source languages and target platforms since different source languages can be translated to the IR, and different target code can be generated from the IR. In terms of abstraction, the IR sits in the middle between source languages and target code. Different IRs can vary greatly depending on their design purposes, such as the general-purpose LLVM [11] for low-level code generation and the domain-specific TameIR [30] for representing MATLAB programs. In order to design proper IRs and generate efficient target code, the compiler research mainly focuses on the multiple levels of IRs and their corresponding compiler optimizations [86].

A compiler needs to support the following code transformation with IRs. Firstly, it *translates* source language code to an IR with only a few optimizations or without any optimizations. This step mainly focuses on generating the correct IR code and passing enough information, such as proper type information. Next, the IR code is *compiled* to target code or another IR code using various compiler optimization techniques. Program analyses are performed on the IR code to collect program information, such

as types, shape, data dependence, and function invocations. If there are multiple IRs, such information can be used to generate the next level of IR code. The multi-level IR design can benefit from different kinds of optimizations, however, it may introduce extra compilation time cost and code complexity. The last level of IR code is compiled with code generation optimizations to generate target code, such as assembly code or C code. After that, the generated target code can be executed in an interpreter, for example, in the Java Virtual Machine for executing optimized Java bytecode [86], or further compiled by a compiler to generate efficient binary code. The IR approach can support different hardware platforms by compiling the IR code to target code that is suitable for the particular hardware. For example, when the target code is C and should run on a CPU exploiting parallelism, then the compiler can compile the IR code to sequential C code annotated with OpenMP directives [9]. This annotated code can then be compiled by a C compiler which supports OpenMP, such as the GCC compiler [2]. Similarly, if we want to use GPUs, we can generate C code with OpenACC directives [8]. This C code can be then compiled by the NVIDIA PGI compiler [7] to binary code for GPUs.

In recent years, IRs were exploited for query processing in order to exploit compiler optimizations. For instance, HyPer [64] translates query execution plans to the low-level IR LLVM which has an efficient compiler back-end. MonetDB has its own IR, called *MAL*, which is then optimized [60]. However, MAL is not compiled to target code but executed in an interpreter. While compiling SQL queries (via IRs) to efficient executable code can have significant performance benefits, compilation and optimization are time-consuming. Thus, the approach might not be beneficial for ad-hoc queries that are only executed once, but promising for query templates that are frequently used.

Chapter 3

Outline of HorsePower

In order to provide a holistic solution for database query processing, we propose a new framework, called *HorsePower*. Its core is an array-based intermediate representation, called *HorseIR*. With HorseIR, the query processing for a column-based in-memory database can be mapped to a set of well-defined array operations. Moreover, additional analytics functions, which are integrated into SQL as user-defined functions (UDFs), can be translated into HorseIR as well. HorsePower is able to provide effective optimizations for array operations in HorseIR programs generated from SQL, analytics functions, or both. The framework also provides a data management system for supplying data during the execution of the generated code.

The framework is depicted in Figure 3.1. At a high level, the framework consists of two major components: *HorseSystem* and *HorseRuntime*. HorseSystem is responsible for generating, optimizing, and executing HorseIR. There are three main components outlined in the following sections: an array-based intermediate representation HorseIR in Section 3.1, a front-end for HorseIR code generation in Section 3.2, and a back-end for HorseIR compilation in Section 3.3. HorseRuntime is a runtime support system for managing table I/O, relations, and metadata, that assists the execution of HorseIR as outlined in Section 3.5.

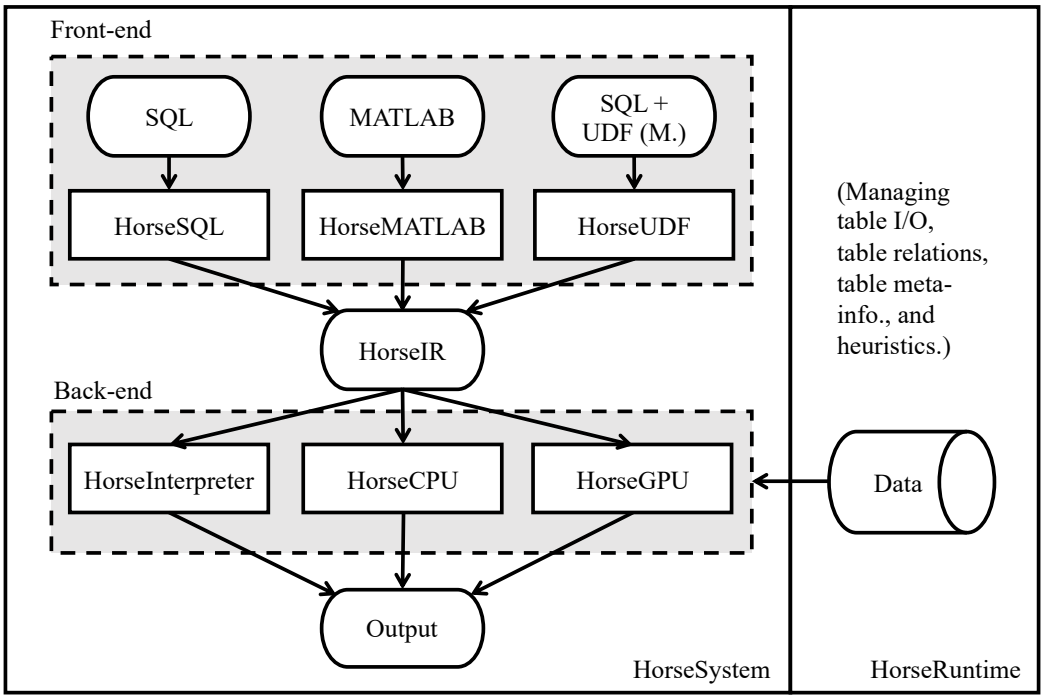


Figure 3.1 – Overview of the HorsePower framework

3.1 HorseIR Design

HorseIR is an array-based high-level intermediate representation, designed for query processing on column-based in-memory relational data. It supports a rich set of types to reflect the many types found in RDBMS. Moreover, it has two kinds of shapes: vector for homogeneous data and list for heterogeneous data. A column in a database can be treated as a vector in HorseIR, and the aggregated data in a query can be treated as a list in HorseIR. In addition, HorseIR provides a set of well-defined built-in functions for many basic array operations. The core relational algebra can be mapped to these functions making the translation from an SQL query to HorseIR possible. The design and implementation of HorseIR can be found in Chapter 4.

3.2 Front-end Design

The HorsePower front-end aims at mapping SQL queries and MATLAB programs to HorseIR, and generating HorseIR code automatically. The front-end consists of the following source-to-source translators: HorseSQL, HorseMATLAB, and HorseUDF. The details of the front-end are presented in Chapter 5.

HorseSQL, introduced in Section 5.1, is a translator designed for generating HorseIR code from SQL queries. We present the mapping from a set of core relational algebra operations to array-based HorseIR operations. From there, we present our strategies for translating full SQL execution plans to HorseIR. Finally, we provide a complete translator for these execution plans.

HorseMATLAB, presented in Section 5.2, is a translator designed for translating MATLAB programs to HorseIR programs. As a popular array-based language used in the field of statistics and engineering, MATLAB is an interesting source language for generating HorseIR programs as they both support array programming. In our solution, we use the McLab framework [5] to first compile MATLAB to optimized TameIR, which is an IR used in McLab. TameIR code is then translated to HorseIR.

HorseUDF, described in Section 5.3, is a translator designed for generating HorseIR from SQL queries with UDFs. MATLAB is a suitable programming language for writing analytical functions. Thus, HorsePower supports UDFs written in MATLAB. Since the SQL queries and the embedded MATLAB functions can be both translated to HorseIR, we can compile all in a holistic manner to target code. This approach has the advantage of reducing the expensive data movement between the analytical and the database system. It also brings more cross-method optimization opportunities as the two kinds of HorseIR code can be optimized together.

3.3 Back-end Design

The HorsePower back-end is responsible for translating HorseIR code into executable code, applying a wide range of optimization techniques during this process. The back-end consists of the following components: HorseInterpreter, HorseCPU, and

HorseGPU. The first provides an interpreter environment, the other two use a compilation approach. They are built with Flex [1] and Bison [3], which are popular tools for building interpreters and compilers. The back-end is described in Chapter 6.

HorseInterpreter is an execution engine providing an interpreter-based execution environment for HorseIR. Many database systems currently do not compile SQL queries into executable code but use an interpreter to execute the operations in execution plans as to avoid the compilation of ad-hoc queries that might execute only once. Thus, we also offer an interpreter environment. HorseInterpreter follows a standard interpreter design: it directly interprets input HorseIR programs, making calls to a built-in library with a rich set of well-defined built-in functions. Interpreting, rather than compiling, is effective when the input data is small and can be processed fast with these efficient built-in functions. Details can be found in Section 6.1.

The compilation environments work in a two-step approach. A HorseIR program is first transformed into C code, a target language, with a whole set of compiler optimizations performed in this code transformation process. In a second step, the C code is then compiled into executable binary code. By using C as our target code, we can fall back to performant C compilers that can take the target hardware into account, leading to a second round of optimizations.

HorseCPU is a compiler back-end generator which translates HorseIR code to executable binary code for CPUs. A HorseIR program is first parsed to generate an abstract syntax tree (AST), and then its type and shape information are propagated to ensure all expressions obey proper type and shape rules before generating internal AST nodes. These AST nodes are analyzed by an optimizer which constructs data dependency graphs for further optimizations. In the optimizer, the optimization of loop fusion plays a crucial role in optimizing array-based HorseIR programs. With loop fusion, the aim is to generate fewer loops in order to reduce intermediate results. Furthermore, database specific patterns are exploited in this optimization phase. Care is taken to minimize interference between fusion-based and pattern-based optimization. After these optimizations, parallel C code is generated. The C code is then compiled using C compilers to generate CPU code. Details can be found in Section 6.2.

HorseGPU is another compiler back-end generator which translates HorseIR code

to executable binary code for both CPUs and GPUs. Since GPUs have a superior performance for complex data computation on massive data with excellent support for data parallelism, the compute-intensive operations of the HorseIR program are translated to GPU code. The other part of the HorseIR program is still mapped to CPU code. The first steps of the transformation of a HorseIR program are similar to HorseCPU, and AST nodes are generated. From there, additional information is analyzed to decide which portion of a program should be offloaded to GPU in order to speed up the overall program. Therefore, it provides customized program analyses and optimizations for generating parallel C code for GPU. Details can be found in Section 6.2.

3.4 Built-in Functions

A HorseIR program may contain many built-in functions. For many of them, HorsePower provides efficient parallel C implementations as a built-in function library. This library improves the execution performance of HorseInterpreter, and reduces the compilation time for HorseCPU and HorseGPU. We present the most important built-in functions and their implementations in Section 6.3. HorseCPU and HorseGPU, during the optimization phase, have to determine whether they use these implementations, in order to better exploit loop-based fusion, emitting C code instead of calling built-in functions from the library.

3.5 Runtime Support

HorseRuntime provides runtime support in managing data and supporting heuristics. HorseIR needs the support of a runtime system during the code execution in order to load data and its metadata, which can be table metadata for columns of tables and the primary/foreign key relationships. Thus, the data management is mainly for (1) loading data into tables; (2) collecting the basic metadata of tables; and (3) supporting possible logs and dumps. Moreover, HorseRuntime is designed for

maintaining a set of heuristics which play an important role in making decisions for determining optimized execution paths at runtime. The input of a heuristic function takes many factors into accounts, including array sizes and types. The details of data management in HorseRuntime can be found in Section 6.4.

Chapter 4
HorseIR: the Core

In this chapter we introduce in detail HorseIR, an array-based intermediate representation. The design of HorseIR aims at representing the execution logic of database queries on main-memory column-based database tables and programs written in the array-based language MATLAB. We first introduce the design principles for HorseIR in Section 4.1. We then present the language features of HorseIR, including program structures in Section 4.2, types in Section 4.3, built-in functions in Section 4.4, and program statements in Section 4.5.

4.1 Introduction and Design Principles

HorseIR is a typed three-address intermediate representation (IR) which supports a set of modern language features, including modules, methods, static fields, local variables, and statements[1]. The design principles of HorseIR are as follows:

High-level IR. With the purpose of supporting effective program analyses for optimization, HorseIR provides a concise set of data structures and a rich set of array-based built-in functions. Data in HorseIR can be either a vector or a list, and operations on the individual elements can be processed without explicit loop iteration. In terms of level of abstractions, HorseIR is a high-level IR

[1] A clean HorseIR language specification can be found in Appendix A.1.

but still at a much lower level than user-facing programming languages, such as MATLAB and SQL. Therefore, the translation from these higher-level languages can follow a clear methodology, and there are plenty of optimization opportunities when HorseIR is translated to target code.

Extensibility. HorseIR follows the conventional design with a set of primitive functions. Source languages may introduce new operations that need to be supported by new functions in HorseIR. Therefore, it is important to have a mechanism in HorseIR that allows the set of built-in functions to be easily extended and optimized.

4.2 Program Structure

In order to ensure code quality, HorseIR follows the standard design for program structures, organizing code into modules, methods, and blocks. As depicted in Figure 4.1, a HorseIR program may consist of multi-level blocks, various methods, and different modules. Modules can be imported to share code with other modules, allowing even relatively complex HorseIR code to be clearly structured. In the rest of this section, we introduce modules in Section 4.2.1, methods in Section 4.2.2, and blocks and scoping in Section 4.2.3. We explain them along the code example depicted in Figure 4.2 which shows an SQL query and a corresponding HorseIR program.

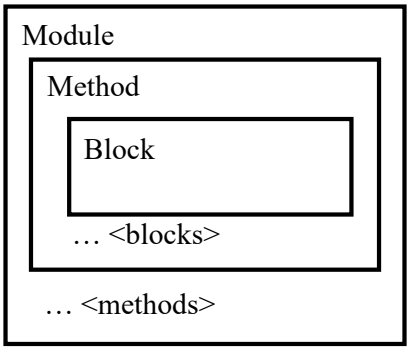


Figure 4.1 – Overview of HorseIR program structure

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

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

Figure 4.2 – Example of a HorseIR module (bottom) for an SQL query (top) which returns the number of stores with relatively big discounts (50-80%).

4.2.1 Modules

A valid HorseIR program consists of a set of modules, with each module defining a collection of imports, static methods, and static fields. For example, Figure 4.2 shows the HorseIR module `BigDiscount`. In addition to named modules, there is a pre-defined module `Builtin`, which implements the basic mathematical and database operations, such as `load_table`, `geq`, `lt`, etc. If a module contains a method called `main`, then this can be used as an entry point of a program. In the `BigDiscount` example module, there is a `main` method that reads from the database table `stores`, loads the column `discount`, and then executes the subsequent statements. Further, a module can import one or more methods from another module using an *import* statement.

With this simple module design, HorseIR provides a mechanism for modularizing complex software and provides a flexible way of specifying reusable libraries, such as the `Builtin` module [68].

A static field is defined in a module as a global variable, and a local variable is defined in a method. Each static field and local variable must have a declared type. A static field may be shared through the directive *import*.

4.2.2 Methods

A method has zero or more parameters and zero or one return value. Parameters are passed by value, which simplifies program analysis, but also means that *copy-elimination* is an important optimization, which is needed to efficiently execute MATLAB [33]. Method calls preceded by the `@` indicate user-defined or library built-in function calls, whereas those without the `@` are HorseIR system functions, such as `check_cast` for checking whether an expression returns a designated type.

4.2.3 Blocks and Scoping

The scope of a declared name needs to be determined statically. A HorseIR program contains the following scopes:

- *Program scope* is a root scope which contains all modules in the compilation unit.

- *Module scope* includes methods and static fields in a module. Content can be declared in any order.

- *Method scope* consists of parameters and local variables in a method. Variables must be declared before use.

- *Block scope* considers blocks defined as part of control-flow structures that define new scopes.

In order to be a valid HorseIR program, declarations within a scope must be unique: (i) a module name in the program scope; (ii) a method name in a module; and (iii) a static field name in a module. In addition, the use of an identifier may conflict with another identifier that needs to be resolved. An order is defined to check conflicts: (1) block scopes if existing; (2) method scope; (3) module scope; and (4) imported content. It is acceptable for local variables to shadow global declarations (i.e., static fields), and for static fields and methods to shadow imported content. That is, if a local variable in a method and a static filed as a global declaration have the same name, the use of the name refers to the local variable within the method.

When importing a module, the imported module content may optionally be used without the module name if they have not been shadowed. For example, in Figure 4.3, the call to the method `foo` in module `A` resolves to the local method `A.foo`. However, another method `bar` resolves to `B.bar` because there is no local method `bar`. In order to call the method `foo` of module `B`, one has to explicitly state `B.foo`. In the case of shadowing, the fully qualified name is required. Both static fields and methods follow the same rules, and local variables cannot be imported. When two imported modules contain an element of the same name, the last import shadows the earlier import.

```
module B {
    def foo() : i32 { ... }
    def bar() : i32 { ... }
}
module A {
    import B.*;
    def foo() : i32 { ... }
    def main() {
        ...
        a:i32 = @foo();    // Resolves to A.foo
        b:i32 = @bar();    // Resolves to B.bar
        c:i32 = @B.foo();  // Resolves to B.foo
        ...
    }
}
```

Figure 4.3 – Method name resolution when importing a module.

4.3 Types

HorseIR provides a rich set of base and advanced types. Deciding on the type system was a very important decision in the design of HorseIR. The key decision was that HorseIR should be statically typed, but with a special wild-card type that allows for the case when a static type is unknown, thus indicating where a static type inference at compile time or a dynamic type check at runtime must be made. This tension between static and dynamic typing is partly due to the fact that database tables have declared types; thus, generating statically-typed HorseIR from queries should be possible, and is preferred. Furthermore, it has been well established that static types and shapes can lead to much more efficient array-based code, therefore one should aim for as much static typing as possible [51]. However, many common array languages, such as MATLAB, are dynamically typed, and by offering a wild-card type, it is possible to generate HorseIR from programs or UDFs written in those languages. We introduce base types and homogeneous arrays in Section 4.3.1, advanced heterogeneous data structures in Section 4.3.2, and a wild-card type in Section 4.3.3.

Table 4.1 – List of HorseIR base types

Name	Alias	Letter	Byte	Description
boolean	bool	B	1	0 (false) and 1 (true)
small	i8	J	1	Half short integer
short	i16	H	2	Short integer
int	i32	I	4	Integer
long	i64	L	8	Long integer
float	f32	F	4	Single precision
double	f64	E	8	Double precision
complex	complex	X	8	Complex numbers (2 floats)
char	char	C	1	Half short integer or char
symbol	sym	Q	8	Symbol, but stored in integer
string	str	S	8	String
month	month	M	4	Month (YYYY-MM)
date	date	D	4	Date (YYYY-MM-DD)
date time	dt	Z	8	Date and time
minute	minute	W	4	Minute (hh:mm)
second	second	V	4	Second (hh:mm:ss)
time	time	T	4	Time (hh:mm:ss.lll)

4.3.1 Base Types and Homogeneous Arrays

HorseIR supports a rich set of base types for arrays as shown in Table 4.1 and includes:

- all the base types typically found in modern languages (boolean, char, short, int, long, float, double, and complex);
- additional base types that are used in SQL (string, month, date, time, datetime, minute, and second); and
- a special type *symbol* as an efficient representation of immutable strings that is critical for query performance.

A base type is a common type shared in a collection of items. If a variable is declared along with a base type, the variable refers to an array of items with this type. If there is only one item, then it is represented as an array of size one. For instance, a vector `(12,25,39):i32` has an integer type with three numbers. The byte size of a single unit of the base type can be found in Table 4.1. Moreover, each type is denoted by a letter which is used to describe type rules in Section 7.2.2.

An underlying principle in array-based programming languages is that many built-in operations are defined over homogeneous arrays. Since each homogeneous array can be stored in a contiguous memory region, it is a cache-friendly design, as well as being easily partitionable for parallelism. Thus, our declarations denote arrays, with each array having an explicit base type, and an implicit extent (number of dimensions) and shape (the size of all dimensions). Only the base type is declared, but the extent and/or shape may sometimes be inferred. For our example in Figure 4.2, the parameter declaration `t1:f64` declares that `t1` is a homogeneous array with a base type of `f64`. Shape inference would be able to determine that it is a vector, based on the output shape of the built-in function `column_value`. We will explain the shape inference in detail in Section 7.2.3.

Table 4.2 – HorseIR advanced types

Name	Alias	Letter	Description
list	list	G	Collection of items
dictionary	dict	N	Key to value
enumeration	enum	Y	Mapping
table	table	A	A table without keys
keyed table	ktable	K	A table with keys

4.3.2 Advanced Heterogeneous Data Structures

Although homogeneous arrays are excellent for core scientific computations, the data stored in an SQL database is not always homogeneous, and may have columns with different data types. Thus, HorseIR offers key heterogeneous data types to effectively capture SQL-like data in a manner that interacts well with array-based primitives. Furthermore, HorseIR supports many important built-in functions for dealing with these data structures, and they are used extensively in the code generation strategies for database queries described in Section 5.1. The collection of the advanced types with heterogeneous data structure include: *list*, *dictionary*, *enumeration*, *table*, and *keyed table*. They are shown in Table 4.2.

Advanced type: list

A list type is an advanced type which provides cells for holding different types. Lists can be nested. Fundamental to list creation is the built-in function `list` which takes an arbitrary number of arguments and returns a list with each argument saved into a single cell. The length of the returned list is the number of cells. The formation of a list type can be described as follows:

```
list_type ::= 'list' '<' cell_type '>'
;
cell_type ::= type { ',' type }
;
type      ::= list_type | <any other type>
;
```

A specific cell type is associated with a list type when all cells of the list have the

same cell type. For example, the variable `t7` in Figure 4.2 is initially associated with a wild-card (`?`) but later inferred as a list type `list<i64>` at compile-time. It stores the value returned from the function `list` that has in this particular example only one input parameter, namely `t5`, which is an array of type `i64` (and in our example this array happens to be of size 1 as it was created by the `sum` primitive). Therefore, the type of the variable `t7`, `list<i64>`, means it is a list which has a set of cells (in our case only one), all of which are homogeneous arrays with a base type of `i64`.

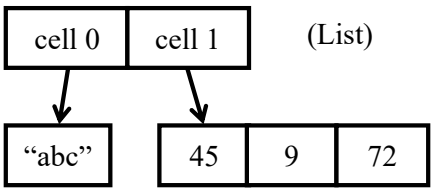


Figure 4.4 – Example of list type

The cells of a list may also contain different types. Figure 4.4 shows an example of a list with two cells holding data of different types, namely a string (`"abc"`) and an array containing three integers (`45,9,92`). This list can be represented as `list<str,i64>` in which its cell types are string and integer types for the two cells, respectively.

A list could be used to represent a database table. In this case, a table could be formed as a list of column names and a list of column values (i.e., the variable `t7`), where the number of column names is equal to the number of columns and all columns agree on their sizes. It is possible that a list can be converted to a vector by using a built-in function `raze` which flattens an input list (including nested lists), creating a vector containing all of the leaf elements of the list. Note that `raze` expects all leaf elements to be of the same type since vectors are homogeneous data structures.

Advanced type: enumeration

An enumeration `enum<T>` takes two input vectors `a` and `b` of the type `T` and indicates for each element `e` in the vector `b` the position in the vector `a` in which `e` occurs. The vector `a` is called the target and `b` is called the source. If there are duplicated items in the target, the first position of the occurrence is returned. In case an item

does not exist in the target, the length of the target is returned.

The example below show two integer vectors `a` and `b`. HorseIR's built-in function `@enum` is used to create an enumeration `ab`.

```
1 a:i32 = (42,7,35):i32;
2 b:i32 = (35,35,42,35,7):i32;
3 ab:enum<i32> = @enum(a, b);
```

As illustrated in Figure 4.5, the variable `ab` first stores the target `a` and the source `b`, in which `a` is a base vector for items in the vector `b`. Then, it stores the indices as a vector of integers `[2,2,0,2,1]` which represents for each element in the vector `b` the position of the item in the target.

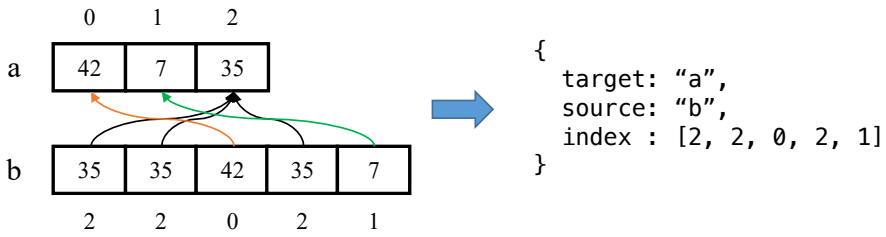


Figure 4.5 – Example of the enumeration type: containing target, source, and index

Advanced type: dictionary

A dictionary manages a list of pairs of keys and values with the type `dict<k,v>`. Each key is associated with a value, and the key and value can be of any type. For example, a key can be a string and the value can be a vector of integers. A valid dictionary implicitly requires that the number of keys must be equal to the number of values. For example, consider the below code snippet:

```
1 symbols:str = ("cityM", "cityT", "cityV"):str;
2 names:str = ("Montreal", "Toronto", "Vancouver"):str;
3 city:dict<str, str> = @dict(symbols, names);
```

The dictionary is created on two string vectors which contain city symbols and names. As a result, the dictionary variable `city` stores the string mappings: {cityM $\rightarrow$ Montreal; cityT $\rightarrow$ Toronto; cityV $\rightarrow$ Vancouver} that enables fast search for city names by given symbol names.

HorseIR supports the built-in functions `keys` and `values` for fetching keys and values from a dictionary. In this example, the keys of the dictionary `city` are stored in the string array `symbols` and the values are stored in the string array `names`. On the other hand, a dictionary can be used to handle the result of a database operation `group by` (details can be found in Section 5.1.1.5) that its key part is a vector of indices while its value part is a list, containing a list of integer vectors.

Advanced type: table and keyed table

A *table* is a special case of a dictionary as shown in Figure 4.6, and it is a HorseIR type to which HorsePower translates database tables. Each key represents a column name, and the associated value is an array representing the corresponding column of the table. Thus, the values of all pairs in the dictionary have the same length, namely the number of rows in the table.

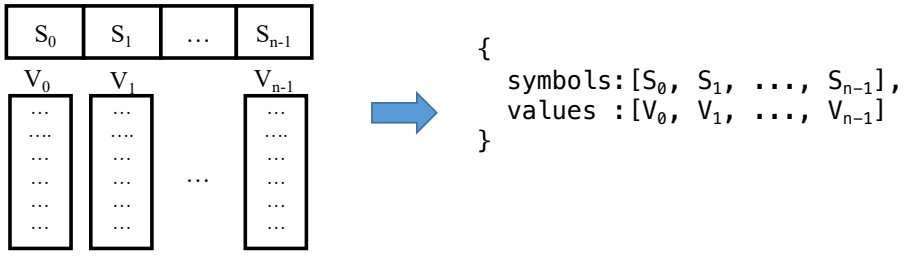


Figure 4.6 – Overview of the table type: containing a list of symbols and vectors

Figure 4.7 presents a *keyed table* which is a table with primary keys. This is distinct from a normal table. A keyed table has a subset of columns as primary keys. Thus, our implementation contains two sub-tables: columns that are part of the primary key are grouped as one sub-table and the other columns are grouped as another sub-table. This design simplifies our implementation by reusing the previous implementation of a normal table. It should be noted that the two sub-tables may contain a different number of columns, and at least one column is required for the sub-table for primary keys.

Figure 4.8 shows the table conversion from a normal table to a keyed table, and vice versa. At least one column from a normal table is selected as the primary key to form a sub-table, and the remaining columns are formed to another sub-table in

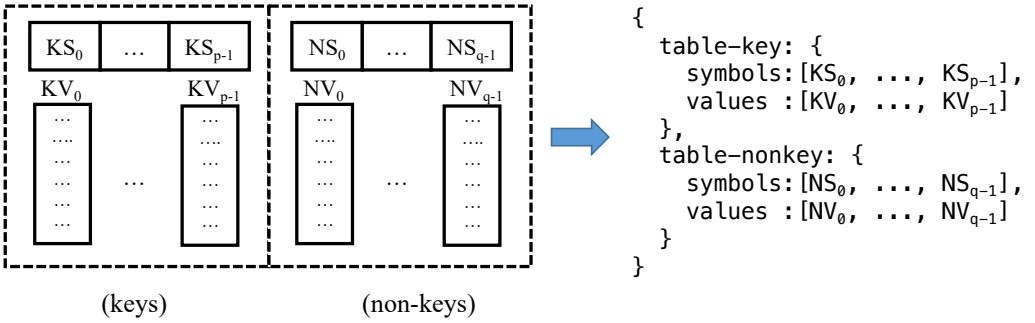


Figure 4.7 – Overview of the keyed table type: containing two normal tables

the keyed table. In our implementation, the cost of the table conversion is minor as this operation avoids actual content movement. Instead, either new list nodes are created for distinguishing two sub-tables: keyed and non-keyed; or a keyed table with two sub-tables are merged to one table. The table columns are then reorganized with their table symbols by passing references.

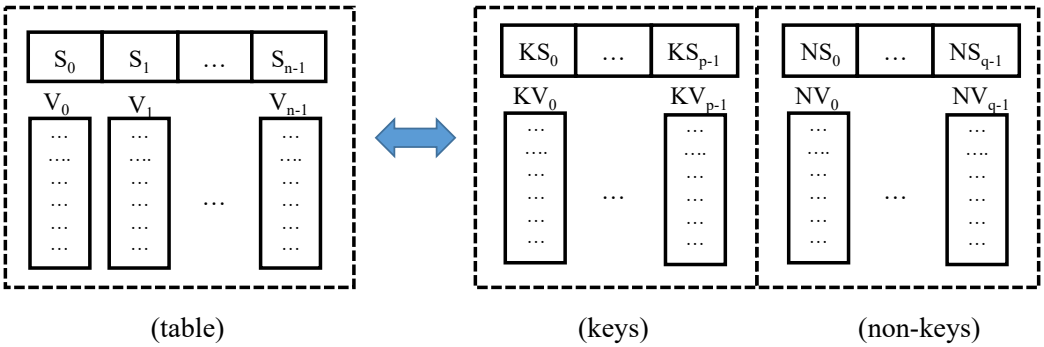


Figure 4.8 – Table conversion between normal and keyed tables

4.3.3 A special type: wild-card

A variable must have a declared type. When its type is unknown, a wild-card (`?`) is assigned. At compile time, the wild-card type is specialized to a concrete type. Figure 4.9 describes how a wild-card type can be specialized showing that a wild-card can represent all possible cases of the aforementioned types. It is possible that

two different wild-card types specialize to the same type. For example, `list<?>` and `list<list<?>>` can result in the same type `list<list<i32>>` when the first wild-card type is eventually assigned to `list<i32>` and the second wild-card type to `i32`. Once a wild-card type is specialized, it is fixed and can no longer be changed. After the phase of type propagation, described in more detail in Section 7.2.2, all wild-card types should have been specialized. Otherwise, this phase fails and terminates the code compilation. A potential optimization to eliminate wild-cards is the type specialization achieved by well-defined type inference. Wild-cards allow HorseIR to handle array programming languages which lack static type information.

```
wild_type     ::= base_type
                | advanced_type
;
base_type     ::= <any base type>
;
cell_type     ::= wild_type {',' wild_type}
;
advanced_type ::= ('list'|'enum'|'dict') '<' cell_type '>'
                | 'table'
                | 'ktable'
;
```

Figure 4.9 – Specialization of a wild-card type with base and advanced types

4.4 Functions

HorseIR provides a rich set of built-in functions (or called primitive functions in array languages) covering vector-based functions, list-based functions, database-related functions, and auxiliary functions. A built-in function is designed to handle various cases when input parameters have different types and shapes. A typical invocation in HorseIR is the symbol `@` with a function name, for example, `@plus`, which represents the function for addition. The complete list of built-in functions can be found in Appendix A.3.

4.4.1 Vector-based Functions

There is a large set of built-in functions for vectors in HorseIR, including element-wise functions, reduction functions, the compress and index functions as introduced in Section 2.2.

4.4.2 List-based Functions

Since HorseIR also includes list-based data structures, it provides a variety of map-like operations. List functions apply a function (`@f`) on cells individually and merge the results into a new list. For example, the `each` function is shown in Figure 4.10.

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

Figure 4.10 – Example of a list function

- **Function each** applies function `@f` to each cell in a list and collects the results in a list. Given the input shape `list<`$L_0, \ldots, L_{n-1}$`>` the output shape is thus `list<@f(`L_0`),...,@f(`L_{n-1}`)>`.
- **Function each_left** takes three parameters: a function, 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<`$L_0, \ldots, L_{n-1}$`>` and `A`, the function produces a new list with shape `list<@f(`L_0, A`),...,@f(`L_{n-1}, A`)>`.
- **Function each_right** takes three parameters: a function, 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<`$L_0, \ldots, L_{n-1}$`>`, the function produces a new list with shape `list<@f(`A, L_0`),...,@f(`A, L_{n-1}`)>`.

- **Function each_item** takes a function and two lists of equal length as input for evaluating 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(`$L_{a(n-1)}, L_{b(n-1)}$`)>`.

As an example of its use, we will see in Section 5.1.1.4 how the built-in function `each_right` is used in the translation of an SQL query with aggregation and groupby to HorseIR code.

4.4.3 Database-related Functions

HorseIR also supports database-related functions specialized for database operations. Some functions are mainly used to simulate how a database manages columns and tables. For example, the function `load_table` loads a database table into a HorseIR table, and the function `table` takes column names in a vector and column values in a list, and returns a table. Moreover, there are a couple of important functions for representing operators in database queries. We describe several of them in detail in Section 5.1, such as `group` for aggregating data into groups, and `join_index` for joining columns with given specific conditions.

4.4.4 Auxiliary Functions

HorseIR provides auxiliary functions that supplement the aforementioned functions. For example, the function `len` takes one argument and returns a one-element vector which indicates the length of the argument: the number of elements in a vector or the number of cells in a list; and the function `range` takes an integer `N` as input returns a vector containing integers from `0` to `N-1` in ascending order.

4.5 Program Statements

A HorseIR method consists of zero or more statements, including expressions, assignment, and control statements.

4.5.1 Expression Statements

An expression statement has only an expression without assigning anything to target variables. It evaluates to a value which can be a function call, a literal or an identifier. This is helpful for functions without return values or with side effects. For example, the function `@print` needs to print a message before returning its integer exit code which sometimes can be discarded without storing the exit code to a variable.

4.5.2 Assignment Statements

An assignment statement consists of left-hand side target variables and a right-hand side expression. In the case of multiple return values, more than one target variable must be present. Target variables may either be declared with their respective types or assigned.

When declaring a new variable in an assignment statement, it is allowed to have a variable declaration with an associated type, for example, `x:i32=<expr>` where the result of the expression is assigned to an integer variable `x`. Variable declaration may also bind variable names to their associated type without an initialization expression. For example, `var x:i32;` where the variable `x` is declared as an integer. The declared new variable can be assigned in later statements.

4.5.3 Control Statements

Control statements exist in programming languages for manipulating code execution. A control statement can be either conditional or unconditional. A conditional control statement includes one or more blocks controlled by conditions, including *if*, *while*, and *repeat* statements. A condition is an expression whose result determines which block will be executed. It can be nested when a conditional control statement contains another conditional control statement within a block. An unconditional control statement simply redirects to a program point without having any conditions. The following control statements are supported:

If-statement allows only one of two blocks to be executed depending on the result of its conditions. The condition result must be a boolean one-element vector.

```
size:i64  = @len(A);
cond1:bool = @eq(size, 1:i64);
if ( cond1 ){
    ...  // execute when size == 1
}
else {
    cond2:bool = @gt(size, 1:i64);
    if ( cond2 ){
        ...  // execute when size > 1
    }
    else {
        ...  // execute when size < 1
    }
}
```

While-statement defines a loop that executes its body until its condition is set to false. The condition value must be a boolean one-element vector.

```
total:i64 = @sum(A);
cond:bool = @gt(total, 100:i64);
while ( cond ){ // enter when cond is true
    ...         // execute
    cond = ...; // update cond
}
```

Repeat-statement is almost the same as the while-statement except the condition result needs to be an integer that controls the specific number of iterations inside the internal loop. It should be noted that the number of iterations is initialized by the value of the condition at the beginning of the first iteration.

```
repeat (100:i64) {
    ...  // repeat 100 times
}
```

Unconditional control statements consist of break, continue, and return statements. Both break and continue statements have to appear inside a loop. A break statement exits a loop, whereas a continue statement jumps to the next iteration. A return statement exits the function and optionally returns a list of values to its calling context.

```
1  while(...){        // enter loop
2      ...
3      if(...){
4          break;     // goto line 10
5      }
6      else {
7          continue;  // goto line 1
8      }
9  }
10 return ...;        // return values
```

Chapter 5
Front-end: Compiling to HorseIR

The front-end of HorsePower is in charge of transforming a program written in a high-level language to a HorseIR program. HorsePower currently supports the transformation for two source languages: SQL for database query processing and MATLAB for array programming. Furthermore, it supports UDFs embedded in SQL statements, as long as they are written in MATLAB. These three components are depicted in Figure 5.1. In Section 5.1, we present HorseSQL which translates SQL queries to HorseIR. Next, in Section 5.2, we show HorseMATLAB which translates stand-alone MATLAB programs to HorseIR. Finally, in Section 5.3, we present HorseUDF which translates SQL statements with embedded UDFs written in MATLAB into HorseIR.

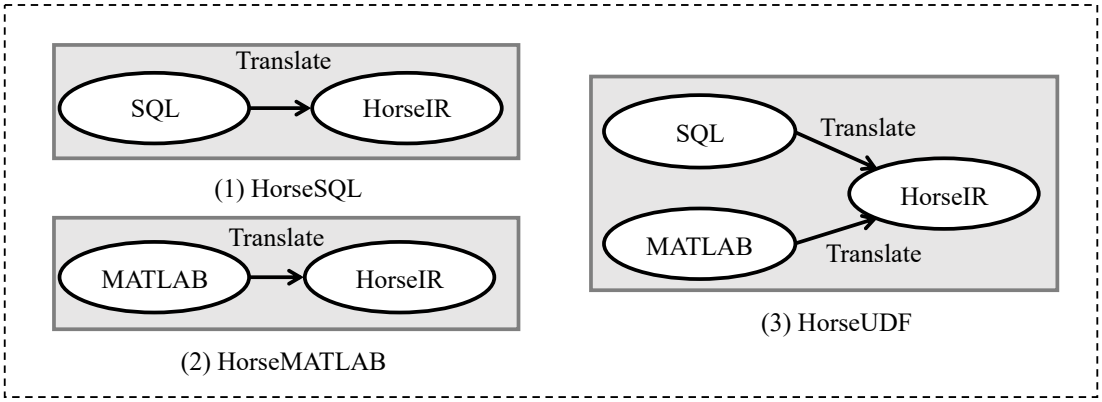


Figure 5.1 – Overview of HorsePower front-end which translates SQL queries, MATLAB functions, and MATLAB UDFs embedded in SQL code into HorseIR programs

5.1 HorseSQL: SQL-to-HorseIR Translator

Code generation from SQL to HorseIR starts with the execution plan of the SQL query. The execution plan consists of a tree of relational operations and is generated by the SQL optimizer of a database system. Query optimizations have been extensively studied in the research literature over the past decades. The generated plans can be considered an optimized way to execute the original SQL queries. Thus, we use these optimized plans instead of the original SQL query as a starting point, in order to take advantage of all the optimizations already developed by the database community. As such, we provide *HorseSQL*, a translator which translates execution plans to HorseIR, and its specification can be found in Appendix B. In this section we show first how the individual relational algebra operators that can be found in such a plan and which have been introduced in Section 2.1.3, are mapped to HorseIR code snippets. We then explain how HorseSQL parses the full execution plan, traverses it, and generates comprehensive HorseIR code.

5.1.1 Mapping Relational Algebra to HorseIR

We have given an overview of relational algebra in Section 2.1.3. Mapping relational algebra operators to HorseIR is more than a one-operator-to-one-built-in-function mapping. Instead, relational operators can sometimes be translated to a sequence of HorseIR functions.

5.1.1.1 Projection

Recall that a projection $\Pi_{c_1,c_2,\ldots,c_n}(R)$ returns a relation only containing the columns c_1 to c_n from R, written in SQL as `SELECT c1,c2,...,cn, FROM R`. Figure 5.2 shows the corresponding operations in HorseIR. As can be seen, the function `column_value` loads a column given a table name. The column names are formed into a string or symbol vector. Then, a new table is returned with the function `table` which takes both column names and values.

This code snippet first loads a table `R` and saves it to the variable `a0`. Columns

```
...
a0:table = @load_table(`R:sym);
c1:f64 = check_cast(@column_value(a0, `c1:sym), f64);
c2:i32 = check_cast(@column_value(a0, `c2:sym), i32);
... // load more columns
cn:sym = check_cast(@column_value(a0, `cn:sym), sym);
table_name:sym = `c1`c2`...`cn:sym;
table_column:list<?> = @list(c1,c2,...,cn);
new_table:table = @table(table_name, table_column);
...
```

Figure 5.2 – Example of projection in HorseIR

`c1, c2, ..., cn` in the table `a0` are loaded. Next, we assign column names to the variable `table_name` and column values to the variable `table_column`. Since columns may have distinct types, we use a list type to indicate its cell types. Finally, a new table is returned after calling the function `@table` with new table names and columns as input.

5.1.1.2 Selection

Recall that a selection $\sigma_P(R)$ returns only tuples from `R` that fullfil condition `P`, where $P = (P_1 < op_1 > \ ... \ < op_{n-1} > P_n)$ with any $op \in \{\wedge, \vee\}$ and P_i conditions on attributes. In SQL, conditions are in the WHERE clause. HorseIR has unary element-wise built-in functions for the typical comparison predicates, such as `@lt` for "<" and `@eq` for "=". They are executed on the full columns and return a boolean vector. Furthermore, HorseIR has two built-in boolean functions `and` or `or`, again returning a boolean vector. As in an SQL query a selection is always in connection with other operators, like projection, the boolean vector of the selection is then input for further processing. For instance, for the statement `SELECT c3 FROM table WHERE c1 < 100 AND c2 = 10`, the corresponding HorseIR code is shown in Figure 5.3. First, the selections on c1 and c2 result in two boolean vectors being input of the `and` function. The final boolean vector mask is then input together with column c3 of the built-in function `compress` which selects from c3 only the elements where the mask vector has a `1`. That is, the function `compress` connects selection and projection.

```
...
t0:bool   = @lt(c1, 100:f64);  // c1 < 100
t1:bool   = @eq(c2, 10:f64);   // c2 = 10
mask:bool = @and(t0, t1);
newC:f64  = @compress(mask, c3);
...
```

Figure 5.3 – Example of selection in HorseIR

5.1.1.3 Join

Recall that a join $R_1 \bowtie_{COND} R_2$ connects the records from table R_1 and R_2 that fullfil the conditions in $COND$.

General joins

When a join is based on conditions between columns of R_1 and R_2 that do not reflect a primary or foreign key relationship, the join can be computed using the built-in function `join_index`. This function supports many kinds of joins, including on one or multiple columns, one or multiple conditions, and same or different types of columns.

```
...
t0:list<i64> = @join_index(@eq, c1, c2);
t1:i64 = @index(t0, 0:i64);  // indices for c1
t2:i64 = @index(t0, 1:i64);  // indices for c2
t3:?   = @index(a, t1);      // rows from a after join
t4:?   = @index(b, t2);      // rows from b after join
...
```

Figure 5.4 – Example of join in HorseIR

Figure 5.4 introduces a HorseIR code snippet for implementing the SQL query (`SELECT R1.a,R2.b FROM R1,R2 WHERE R1.c1=R2.c2;`) using the function `join_index` to perform an equal join on the two columns `c1` and `c2` and then retrieving the proper elements from columns `a` and `b` for projection. The implementation is depicted in Figure 5.5. Both input integer arrays are on integers and the output index consists of a list of two arrays of the same length. For instance, the first item (row 0) in the array `c1` is 17, which can be found in the array `c2` two times with the row indices 1

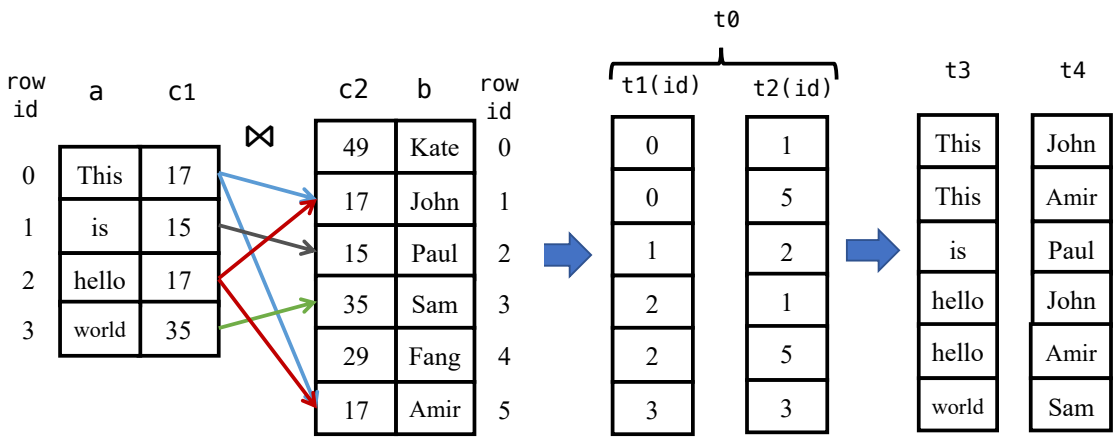


Figure 5.5 – Illustration of the equal join operation in Figure 5.4

and 5. This results in two pairs of indices (0,1) and (0,5) in the result list `t0`, being the first two elements in the output arrays t1 and t2. The integer indices stored in t1 and t2 are later used to fetch the actual rows after the join, for example from arrays `a` and `b` in our example code.

Note that the input parameters for the join, `c1` and `c2` in our example, can be vectors for a one-column join or lists for a multiple-column join. In the second case, the two lists must have the same number of arrays to match. A single join operator `@eq` is for one or more columns, and different join operators for different columns need to be explicitly defined; for example `@eq@neq` means the first column of each of the lists will be connected with an equal condition and the second column with a non-equal condition.

Enumeration as foreign keys

A special case for join in HorseIR is when two tables are joined with an equijoin connecting the primary key in the table with the corresponding foreign key in the other table. For that case, we exploit the primitive data type *enumeration*, introduced in Section 4.3. We use the enumeration to maintain the mapping between primary key and foreign keys by storing the index of the value in the primary column as part of the value in the foreign key column. An example of enumeration for primary key and foreign key mapping is depicted in Figure 5.6. The column `deptid` is the primary key

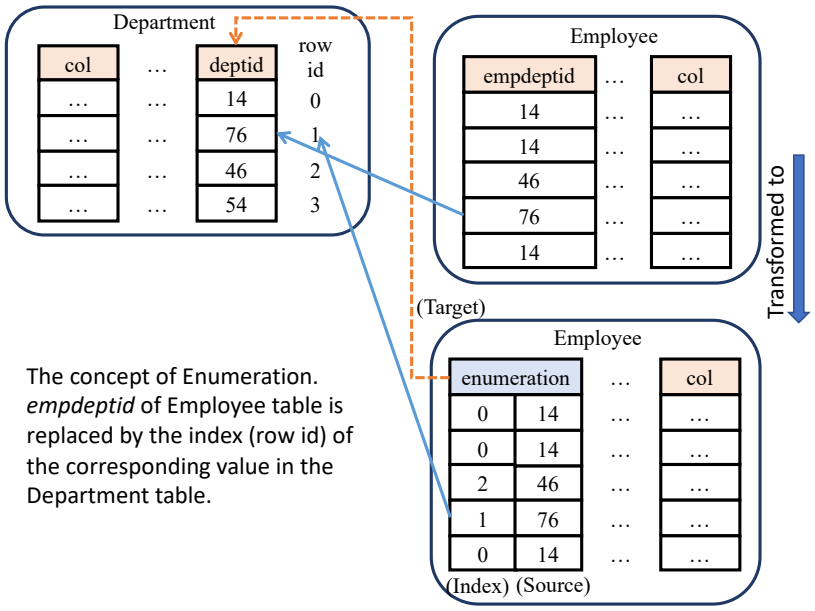


Figure 5.6 – Example of an enumeration in a join with a pair of primary key (`deptid` in the table `Department`) and foreign key (`empdeptid` in the table `Employee`)

in the table `Department`. The row id is simply the row index of a value in the `deptid` column. The column `empdeptid` in the table `Employee` is a foreign key referencing the department in which the employee works. Using enumeration, we transform the table `Employee` and replace its column `empdeptid`. We do this transformation at table storage time. That is, in a pre-processing step, the original `Employee` table is transformed to one that contains the enumeration. The transformed table has no longer an `empdeptid` column, but an enumeration that contains the original column `empdeptid` as a source, the column `deptid` of the table `Department` as a target, and an index array of the same length as the source `empdeptid`, pointing to the corresponding row in `deptid`.

Figure 5.7 shows how HorseIR uses this enumeration format. It first loads the column `empdeptid` from the table `Employee`, which now has an enumeration type `enum<i64>`, into the variable `dept`. Then, the HorseIR function `@values` returns the row id stored in the index component of `dept`, and another function `@fetch` returns the source column `empdeptid`. Since both values are computed during the

```
1 ...
2 emp:table = @load_table(`Employee:sym);
3 dept:? = check_cast(@column_value(emp,`empdeptid:sym), enum<i64>);
4 dept_id:i64 = @values(dept);
5 dept_val:i64 = @fetch(dept);
6 ...
```

Figure 5.7 – Example of enumeration in HorseIR

transformation phase, the cost of invoking the two functions is small during the execution of the HorseIR code. This is for a query `SELECT deptname, empname from Department, Employee WHERE deptid = empdeptid`. We can create the result column for department names by calling `@index(deptname, dept_id)`.

5.1.1.4 Aggregation

Expressing a simple aggregation, such as `SELECT SUM(revenue) FROM table`, is easy in HorseIR because it has equivalent reduction functions. For example, the aggregation `SUM` can be represented as HorseIR function `@sum`, which takes an array as input and returns an array of one element with the total value. Other reduction functions, such as MIN, MAX, and COUNT, have their corresponding HorseIR functions, for instance, `@min`, `@max`, and `@count` respectively.

5.1.1.5 Group By

The clause *GROUP BY* in SQL is designed for grouping one or more columns. In order to express this, HorseIR provides a built-in function, called `@group`, which takes an array or a list as input, and groups together the indices (positions) of the array or list that have the same value. It returns a dictionary that there are as many keys as there different values.

Figure 5.8 presents an example of the group operation with a single integer column `c1` as input, and a dictionary as output (i.e., dict<i64, list<i64>>). The dictionary stores indices instead of the actual values from the input column as keys. Each first occurrence of a unique value is stored as a key, and the value part is a list consisting

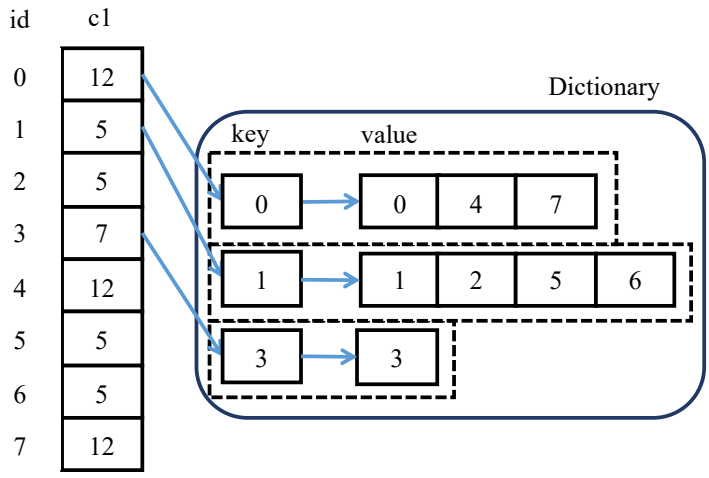


Figure 5.8 – Illustration of the group operation in HorseIR

of all the positions with the same value. That is the key and the first element of the list have the same value.

```
SELECT SUM(c0)
FROM example_table
GROUP BY c1;
```

```
...
t_list:list<i64>            = @list(c1);
t_dict:dict<i64, list<i64>> = @group(t_list);
t_index:list<i64>           = @values(t_dict);
t_val:list<i64>             = @each_right(@index, c0, t_index);
t_sum:list<i64>             = @each(@sum, t_val);
t_vector:i64                = @raze(t_sum);
...
```

Figure 5.9 – Example of an SQL query for group by (top) and its HorseIR code (bottom)

An example for `group by` can be found in Figure 5.9 having an SQL query and its HorseIR code. This example uses the value part of the dictionary. By storing the first position of each value in the vector `c1` in the key, we can quickly retrieve the different values. Grouping all positions with the same value in the values part is needed when the clause GROUP BY is used together with aggregating on a different column, as

shown in our example.

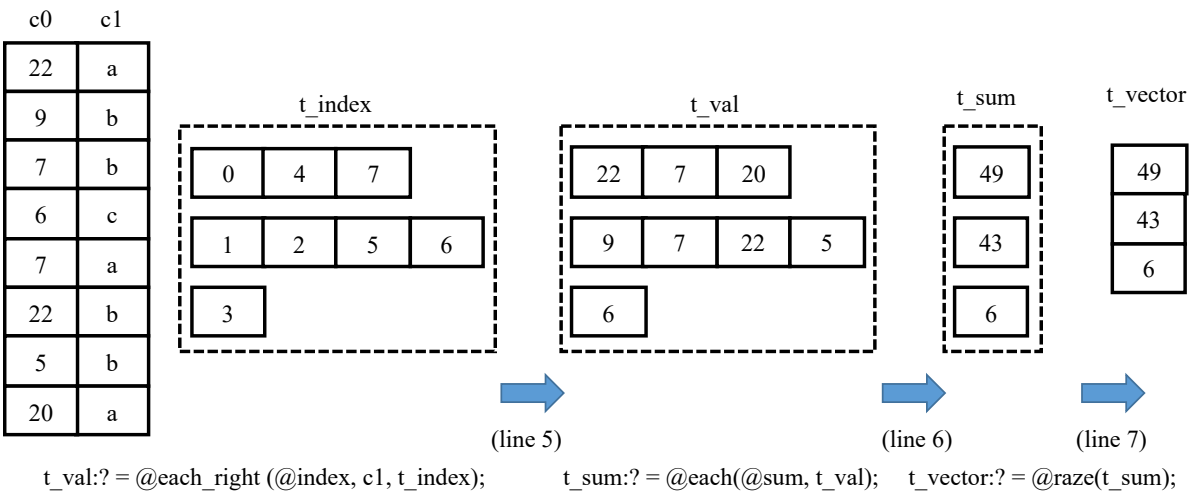


Figure 5.10 – Illustration of the HorseIR code in Figure 5.9

As depicted in Figure 5.10, the value part of the result dictionary (from line 4) works with a list-based function `each_right` to perform array indexing on the vector `c0` (line 5), and calculate the sum in each group using the function `@sum` for each vector in the value list (line 6). The return list can be flattened with the function `raze` and a vector is returned (line 7).

5.1.1.6 Order By

The clause *ORDER BY* in SQL is for sorting a table based on one or more columns. It can be expressed in HorseIR code by using built-in functions `@order` and `@index`.

Figure 5.11 shows an example of the order by operation on the column `c1` from a two-column table. As illustrated in Figure 5.12, it first sorts the column `c1` with the function `@order` and returns an index with the row ids of the sorted values. Then, it fetches the tuples with the function `@index`, including the column `c2`, based on the given row ids and returns a new table as result.

```
SELECT c1,c2 FROM example_table ORDER BY c1 ASC;
```

```
...
t_index:i64 = @order(c1);
t_c1:i64    = @index(c1, t_index);
t_c2:str    = @index(c2, t_index);
...
```

Figure 5.11 – Example of an SQL query for order by (top) and its HorseIR code (bottom)

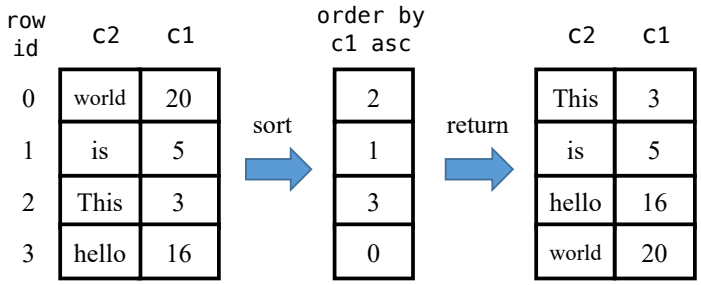


Figure 5.12 – Illustration of the order operation in HorseIR

5.1.2 Code Generation Strategy

In this section we show how a full execution plan is translated to HorseIR code. An execution plan has a clearly defined order of operators. As we have just seen in Section 5.1.1, each of the operators in an execution plan can be mapped into one or more lines of HorseIR code. Our code generation is similar in concept to a compiler back-end with code patterns. Therefore, translating an execution plan into a HorseIR program is straightforward. Our translator supports the execution plans from two database systems: HyPer [64] and MonetDB [40].[1]

HyPer generates optimized execution plans which are expressed as JSON objects. However, its interface does not support UDFs. Thus, when there are UDFs in SQL queries, we use MonetDB's execution plans because MonetDB supports UDFs and its execution plans contain the relevant UDF information, such as function names,

[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

parameters and parameter types. As MonetDB's execution plans follow a traditional tree structure, HorsePower transforms this tree structure to a JSON object, which can then be translated to HorseIR with the infrastructure for HyPer's execution plans with some extensions to handle UDF information. We describe these extensions in Section 5.3.

We introduce our code generation strategy from an execution plan to a HorseIR program as follows:

Traversal strategy. An optimized execution plan has a nested tree-based structure (in the case of HyPer, it is in JSON format). Our code generator traverses the tree using depth-first search (i.e., from leaf nodes up to finally the root), generating code for each operator node. The code for each child is generated, and then the results passed to the next operator. Finally, the root operator returns the code for generating the result table.

Environment objects. Recall that the result of each operator is, in principle, a table. Thus, we designed an environment object which plays an important role in our code generation strategy. It stores the information of intermediate results (i.e., the intermediate tables) after the code generation on each operator. Then, it is passed to the parent operator. When an operator receives all environment objects from all its children, it operates on these objects with the current operation. In our design, the object is similar to a table, but has some additional elements for the purpose of delivering more information for optimizations at this translation level. Therefore, the object consists of:

- `table_name`: a string for a table name (a unique name is assigned for a temporary table);
- `cols_names`: a vector of strings for column names;
- `cols_alias`: a vector of strings for variables to represent the corresponding columns (it has the same length as `cols_names`);
- `cols_types`: a vector of strings for the corresponding variable types (it has the same length as `cols_alias`);

- `mask`: a variable used for boolean selection (or no boolean selection if its value is empty or null);
- `mask_a`: a vector of variable names representing the corresponding columns after boolean selection (it has the same length as `cols_alias`).

Expressions and types. Operator nodes may have expressions as their children nodes. Some expressions, such as the less than (`lt`), correspond directly to one HorseIR statement, whereas others, such as "`between`" require generating several HorseIR statements.

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

Function generation. Each operator in a plan can be mapped to one or multiple HorseIR built-in functions. Relational algebra represented in HorseIR has been discussed in Section 5.1.1. In addition, other operators are supported in HorseIR as well, such as the SQL case statement shown below.

```
CASE
    WHEN <condition> THEN TrueResult
    ELSE FalseResult
END;
```

Instead of generating an `if-else` statement, this statement is translated into a vector form: (a) it first evaluates the condition for all cases and stores the result into a boolean vector `BoolVec`; (b) it computes the true result as `BoolVec * TrueResult`, and the false result as `(~ BoolVec) * FalseResult`; and (c) it sums up the two results and returns the final result.

```
BoolVec = eval(condition) ;
result  = BoolVec * TrueResult + (~ BoolVec) * FalseResult ;
```

The previous pseudo-code can be further optimized if `TrueResult` and `FalseResult` are a combination of 1 and 0, or 0 and 1.

```
BoolVec = eval(condition) ;
if (TrueResult == 1 and FalseResult == 0){
    result = BoolVec  ;
}
else if (TrueResult == 0 and FalseResult == 1){
    result  = ~BoolVec ;
}
```

It should be noted that the values of `TrueResult` and `FalseResult` can be two single numbers that can be either constant or the result of expressions.

5.1.3 Optimizations in Generating HorseIR Code

Our HorseIR code generation is likely to produce redundant or inefficient code. Therefore, we identify and implement the following optimizations in code transformation.

Eliminating dead code. HorseIR code is generated assuming that all results will be needed. However, the final results table may contain only some columns, and thus we only need to retain the code needed to compute those columns. We use a backward slice [84] from the final result to identify the code that needs to be retained, and eliminate all other code, as it is effectively dead code. This is common in the execution plans generated from both HyPer and MonetDB.

Using code patterns for expensive database operations. Expensive database operations, such as `join` and `groupby`, are common in SQL queries. It is critical for performance to avoid large intermediate results between these operations whenever possible. When generating HorseIR code from execution plans, we implement optimizations using code patterns for such database operations. A code pattern is prepared for a compiler that can recognize it within a code snippet, then rewrite it into an equivalent but more efficient code. For example, assume a self-join, which is a special case of the join operation when a table is joined with itself, often on different columns. An example would be joining an `employee` table with itself over the columns `employeeid` and `managerid` to

find who is the manager of an employee. When only aggregation operations occur right after an inner self-join, we replace the join with a `groupby` which is cheaper than join.

Implementing join transformation. Join transformation is used when a primary/foreign key join is pre-computed as discussed in Section 5.1.1.3 but we cannot directly use it because there is first a selection we can still take advantage of it in the following scenarios:

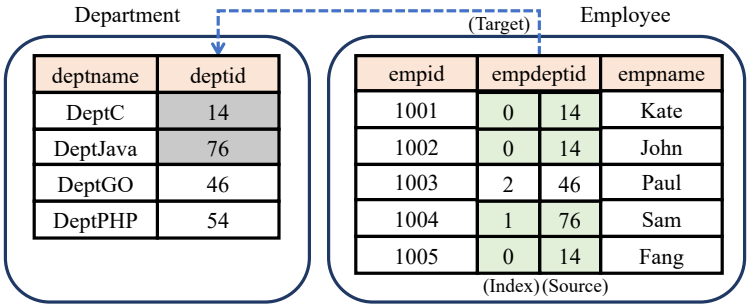


Figure 5.13 – Example of the join transformation: scenario 1

(Scenario 1.) Figure 5.13 illustrates this example with two concrete tables: `Employee` and `Department`, in which the column `deptid` is a primary key and the column `empdeptid` is a foreign key. As discussed in Section 5.1.1.3, the indices of `deptid` are already pre-computed in the `Employee` table.

Figure 5.14 shows an SQL query for this scenario and the corresponding HorseIR program. There is a selection on department name selecting only the first two columns. That is, there is a selection on the table with the primary key. The selection generates the boolean mask `t_dept_mask=(1,1,0,1,1)`. By now deploying the index function on the index vector in `empdeptid`, we can get the `Employee` rows that qualify (depicted in Figure 5.13). From there we can get the corresponding employee names through a `compress` function. The corresponding department names can be fetched by first deploying a boolean selection to get the corresponding row ids, and then applying the indexing function to get the actual values.

```
SELECT deptname, empname
FROM Department, Employee
WHERE deptid = empdeptid AND
      deptname = "DeptC" AND deptname = "DeptJava";
```

```
...
t_dept_mask:? = @member(deptname, ("DeptC","DeptJava"):str);
t_emp_index:? = @values(empdeptid);
t_emp_mask:?  = @index(t_dept_mask, t_emp_index);
t_emp_name:?  = @compress(t_emp_mask, empname);  // empname
t_dept_index:?= @compress(t_emp_mask, t_emp_index);
t_dept_name:? = @index(deptname, t_dept_index);  // deptname
...
```

Figure 5.14 – Example of an SQL query (top) and its HorseIR code (bottom) for the scenario 1

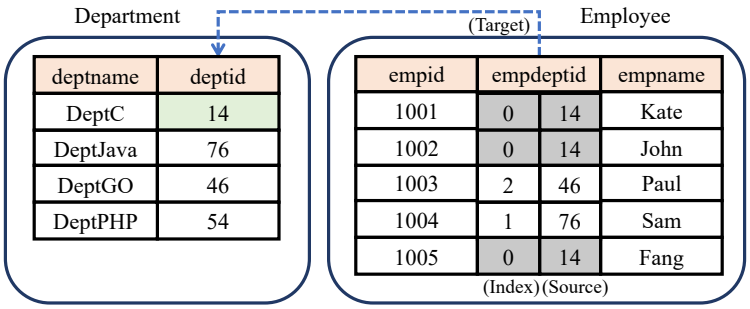


Figure 5.15 – Example of the join transformation: scenario 2

(Scenario 2.) Figure 5.15 shows the scenario when a foreign key is selected with conditions and its key stays the same. The join can be computed in the following steps: (1) we first apply the conditions in the WHERE clause on the table `Employee` to get a mask vector `(1,1,0,0,1)`; (2) we then utilize the index information stored in the enumeration `empdeptid` to probe qualified rows in the table `Department`; (3) we finally obtain employee names using the boolean mask vector, and the department names based on the updated row information from the enumeration (i.e., `(0,0,0)`).

```
1 SELECT deptname, empname
2 FROM Department, Employee
3 WHERE deptid = empdeptid AND
4       empid = 1001 AND empid = 1002 AND empid = 1005;
```

```
1 ...
2 t_emp_mask:?   = @member(empid, (1001,1002,1003):i64);
3 t_emp_index:?  = @values(empdeptid);
4 t_emp_name:?   = @compress(t_emp_mask, empname);   // empname
5 t_dept_index:? = @compress(t_emp_mask, t_emp_index);
6 t_dept_name:?  = @index(deptname, t_dept_index);   // deptname
7 ...
```

Figure 5.16 – Example of an SQL query (top) and its HorseIR code (bottom) for the scenario 2

Figure 5.16 shows the corresponding SQL query and its HorseIR code. We first compute the mask in the table Employee (line 2) and fetch the index information from the enumeration (line 3). We then use the mask to fetch employee names (line 4) and department indices (line 5). Later, we retrieve department names as the result of array indexing with the given indices (line 6).

(Scenario 3.) Figure 5.17 shows the scenario when a key and its foreign key both are selected. By given the selected rows in the two tables, we need to find the rows after an equijoin. The foreign-key column `empdeptid` is first updated to get the qualified rows. Then, the information is passed to the primary-key column `deptid` using the index information stored in the enumeration. Next, we can know the first two rows should be selected while the primary-key column only

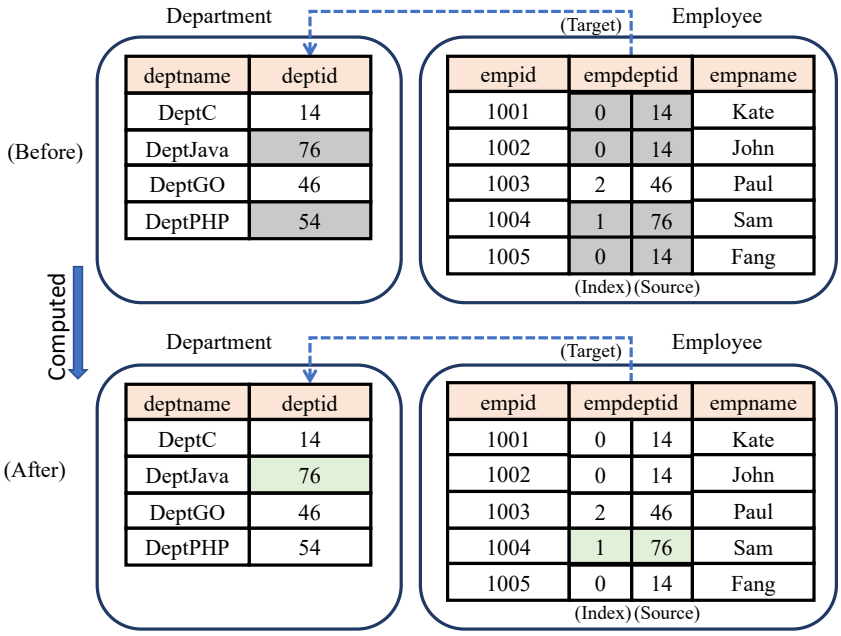


Figure 5.17 – Example of the join transformation: scenario 3

needs the 2nd and 4th rows. Thus, only the 2nd row should be returned. Finally, the table Employee selects the 4nd row based on the updated information, and the table Department selects the 2nd row.

This SQL query and its HorseIR code are depicted in Figure 5.18 that both the primary-key and the foreign-key columns are applied with conditions to select qualified rows after join. We first compute boolean masks for both tables: `t_emp_mask` and `t_dept_mask` (line 2 and 3) We then pass the mask information from the table Department to Employee using the enumeration stored in `empdeptid` (line 4 and 5). We next get a new mask for the table Employee and the updated indices for the table Department (line 6 and 7). Finally, the result of department names and employee names can be fetched using this mask and indices (line 8 and 9).

```
SELECT deptname, empname
FROM Department, Employee
WHERE deptid = empdeptid AND
      deptname = "DeptJava" AND deptname = "DeptPHP" AND
      empid <> 1003;
```

```
...
t_emp_mask_1:? = @neq(empid, 1003:i64);
t_dept_mask:?  = @member(deptname, ("DeptC","DeptPHP"):str);
t_emp_index:?  = @values(empdeptid);
t_emp_mask_2:? = @index(t_dept_mask, t_emp_index);
t_emp_mask:?   = @and(t_emp_mask_1, t_emp_mask_2);
t_emp_index_new:? = @compress(t_emp_mask, t_emp_index);
t_empname:?    = @compress(t_emp_mask, empname);     // empname
t_deptname:?   = @index(deptname, t_emp_index_new);  // deptname
...
```

Figure 5.18 – Example of an SQL query (top) and its HorseIR code (bottom) for the scenario 3

5.2 HorseMATLAB: MATLAB-to-HorseIR Translator

In this section we discuss how programs written in MATLAB can be translated to HorseIR by the HorseMATLAB component.

5.2.1 Mapping MATLAB to HorseIR

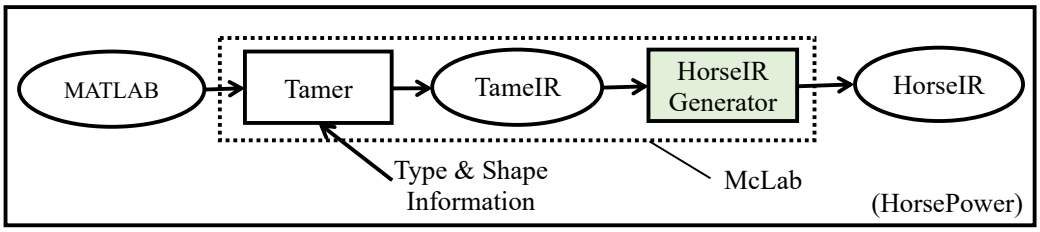


Figure 5.19 – Generating HorseIR code from MATLAB within the McLab framework

Since MATLAB is a sophisticated dynamic language which provides numerous flexible language features, it is challenging to build a MATLAB compiler from scratch. Thus, we employ the McLab framework [5] which provides a complete solution for

parsing, analyzing, and optimizing MATLAB programs, and generating target code as part of HorseMATLAB. Figure 5.19 shows the full workflow for compiling MATLAB to HorseIR. The McLab framework translates MATLAB programs to its own internal IR, called TameIR, which was specifically designed to enable optimization of MATLAB programs. TameIR can be easily translated into various other programming languages to build efficient executable code [55, 51]. Thus, we extend the McLab framework to include a HorseIR generator that can translate TameIR to HorseIR code.

The translation from MATLAB to TameIR, performed by the Tamer module, has to handle MATLAB's many dynamic features and the lack of strict typing. When analyzing the program, the first set of type and shape information is derived from the parameters of the entry MATLAB function. This information is then used to derive the type and shape information for any further variables computed by the statements in the rest of the program.

From there, classic program analysis and optimizations are performed, including interprocedural value analysis, constant propagation and common subexpression elimination to produce as output optimized TameIR code [30]. TameIR can represent MATLAB's matrix and high-dimension arrays, and currently supports an essential subset of MATLAB array operations, such as matrix multiplication and inverse.

However, compared to HorseIR, TameIR lacks support for advanced types that are needed for query executions, such as the `table` data type. Furthermore, it does not provide the database-related functions that HorseIR supports. Thus, in order to merge MATLAB programs and SQL queries, as shown in Section 5.3, we have to further translate TameIR to HorseIR. As both TameIR and HorseIR are array-based IRs, the translation is relatively straightforward.

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

Functions. When compiling multiple MATLAB files, the first function is considered an entry function, i.e., the main function. Since both MATLAB and HorseIR

support the pass-by-value parameter passing, the code generation is straightforward for parameters. However, HorseIR provides an optimization over the default pass-by-value approach by internally employing a copy-on-write mechanism. In this approach, if it is determined that the function does not modify a parameter that is passed to it, then such a parameter is provided through pass-by-reference, saving any overhead associated with making data copies.

Control structures. The common control structures `if-else` and `while` are supported by both languages. When testing a condition in a control structure, the result must be a single boolean element which can be a one-element vector. While in MATLAB a condition check is also considered `true` when a conditional expression evaluates to a non-empty set of elements, this is not allowed for MATLAB programs that need to be translated to HorseIR as this is currently not supported in HorseIR.

Arrays. We support MATLAB programs in an array programming style without using the `for-loop` construct for loop iterations in MATLAB. Instead, programs can use the MATLAB built-in functions which can operate on whole vectors. We have already seen the functions `@compress` and `@index` in HorseIR. In MATLAB, there is one function for this and vector types determine whether we have to translate to a compress or index function. More precisely, given the MATLAB code `A(I)` while both `A` and `I` are arrays, if `I` is an array of boolean values of the same length as `A`, then this is called logical indexing and equivalent to the function `@compress` in HorseIR. If $I = (i_0, i_1, ...)$ is a vector of integer indices, it can be represented with the built-in function `@index` in HorseIR, and results in a new vector $(A[i_0], A[i_1], ...)$ whose length is equal to that of `I`.

Types. As can be seen in Table 5.1, MATLAB has support for a rich set of types, that are supported by the McLab framework. As HorseIR supports many types, it is easy to find a mapping for each, including boolean, character, integer, float, and double. For example, the floating-point value `double` in McLab is translated to `f64` in HorseIR. On the other hand, MATLAB and HorseIR have different

MATLAB Types	HorseIR Types
logical	bool
char	char
single	f32
double	f64
int8	i8
int16	i16
int32	i32
int64	i64

Table 5.1 – Type mapping from MATLAB to HorseIR

type rules in some scenarios. For instance, *integer* + *double* returns integer in MATLAB, but *double* in HorseIR. Therefore, we put restrictions on the type rules to ensure the correctness of the code transformation. For example, the same types are required for the input parameters of the function `plus`.

Shapes. Obviously, we support MATLAB arrays as they are essential components for HorseIR to work on table columns. An array in MATLAB can be either a `1-by-N` or `N-by-1` matrix where `N` is a positive integer greater than `1`. We support the `1-by-N` vector as its data layout is more cache-friendly in MATLAB.

5.2.2 Example Code

Given two points (x_0, y_0) and (x_1, y_1) in a two-dimensional coordinate system, the distance between the two points can be calculated using the following equation:

$$calcDistance(x_0, y_0, x_1, y_1) \;=\; \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

This equation can be implemented as MATLAB code in Figure 5.20. We can see the MATLAB function `calcDistance` takes four parameters (`x0`,`y0`, `x1`, `y1`) as input and returns one variable `z`. All numbers should be floating-point though implicit type information is not required in MATLAB because MATLAB is able to check types at runtime.

```
1 % x0, x1, y0, y1: f64
2 function [z] = calcDistance(x0, y0, x1, y1)
3     z = ((x0 - x1) .^ 2 + (y0 - y1) .^ 2) .^ 0.5
4 end
```

Figure 5.20 – Example of MATLAB

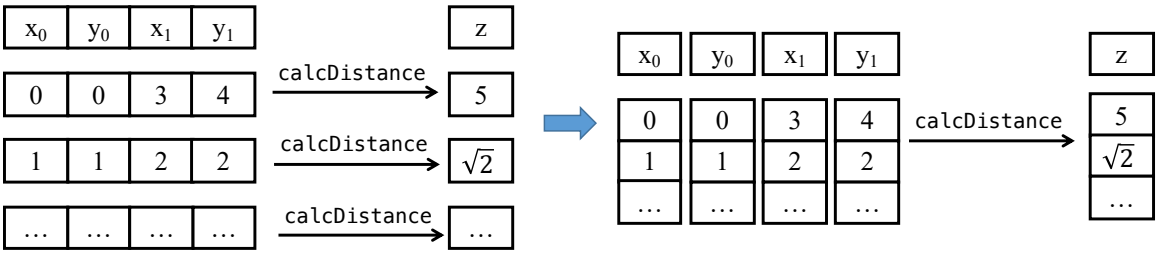


Figure 5.21 – Example of the MATLAB function with vectors as input

Figure 5.21 shows the execution on scalars on the left and on vectors on the right. If the input to the function are scalars, the function simply applies the formula using the scalar values. Thus, when giving the input (0,0,3,4), MATLAB will return the result 5. With multiple sets of input values, the function needs to be invoked for each of them. On the other hand, this program is also able to take arrays as input, as long as they all have the same length, because the built-in functions used in the program (i.e., plus (+), minus (-), and power (.∧)) are element-wise functions that can work on arrays. Here the calculation will be applied for each index of the arrays, and the output is again an array.

Type and shape for parameters. Since explicit type and shape information are not required in MATLAB source code, we need to supply this information to the McLab framework for compiling MATLAB code. Different input type and shape information may lead to different target code. In this example, we define input type and shape information as follows

```
x0: DOUBLE&1*?&REAL
y0: DOUBLE&1*?&REAL
x1: DOUBLE&1*?&REAL
y1: DOUBLE&1*?&REAL
```

where, `DOUBLE` means the double-precision numeric data type which is the default type in MATLAB; `1*?` is the shape of a vector with 1-by-N dimension; and `REAL` defines the input should be real numbers but not complex numbers.

Type and shape propagation. After providing the type and shape information of input parameters, the McLab framework propagates this information and generates its intermediate TameIR code. The annotated TameIR code with type and shape information is shown below:

```
1 function [z:(f64,1*?)] = distance(x0:(f64,1*?), y0:(f64,1*?), x1
    :(f64,1*?), y1:(f64,1*?))
2     mc_t0:(f64,1*?) = minus(x0, x1);
3     mc_t1:(f64,1*?) = power(mc_t0, 2:(f64,1*1));
4     mc_t2:(f64,1*?) = minus(y0, y1);
5     mc_t3:(f64,1*?) = power(mc_t2, 2:(f64,1*1));
6     mc_t4:(f64,1*?) = plus(mc_t1, mc_t3);
7     z = power(mc_t4, 0.5:(f64,1*1));
8 end
```

As can be seen, type and shape information are propagated properly from the input parameters to the return variable. Each variable in this program is associated with the type and shape information. If any type or shape rule violation occurs, the propagation process will be terminated with an error message.

HorseIR code generation. After the intermediate TameIR is generated and stored as TameIR nodes in the McLab framework, a customized code generator translates these TameIR nodes to the HorseIR code below:

```
1 def distance(x0:f64, y0:f64, x1:f64, y1:f64) : f64 {
2     t0:f64 = @minus(x0, x1);
3     t1:f64 = @power(t0, 2:f64);
4     t2:f64 = @minus(y0, y1);
5     t3:f64 = @power(t2, 2:f64);
6     t4:f64 = @plus(t1, t3);
7     z:f64  = @power(t4, 0.5:f64);
8     return z; }
```

5.3 HorseUDF: UDF-to-HorseIR Translator

HorsePower supports SQL queries with embedded UDFs written in MATLAB through its HorseUDF module. For that purpose and as discussed in Section 5.1.2, we use the MonetDB execution plans to generate HorseIR code as the information of UDF calls has been included in these plans. While MATLAB is currently not in the list of languages supported by MonetDB for its UDF implementations, this is irrelevant for generating the execution plans as only the hooks into the UDFs with their input and output parameters are relevant. This is because MonetDB treats all UDF execution environments as a black-box and generates the same execution plan, independent of the language of the UDF implementation. Thus, we can directly use MonetDB's execution plan generator on SQL statements extended with UDFs, independently on in which language the function is actually written. From there, we use the plan-to-HorseIR translator for MonetDB's execution plans, as introduced in Section 5.1, to first generate HorseIR code from the plan with place holders for UDF method invocations. That is, in the HorseIR code, the invocation of the UDF is simply translated into a method invocation in HorseIR. Next, we generate a separate piece of HorseIR code by translating the UDF written in MATLAB using the MATLAB-to-HorseIR translator introduced in Section 5.2. Finally, the two segments of code for SQL and UDFs are integrated into a single HorseIR program.

As discussed in Section 2.1.5, our current implementation supports two kinds of UDFs, namely *scalar* and *table* UDFs. In order to conform the MATLAB functions to the semantic form expected of these types of UDFs, we enforce the following restrictions on the implementation of such MATLAB functions. (1) A function must have one return statement with either a single vector (for scalar UDFs) or a table-like data structure (for table UDFs). (2) Executing the MATLAB function individually on each scalar element of an array and returning the result set as an array should be equivalent to executing the function on the full array as input (that is, f($\{x_1$,x_2,..,$x_n\}$) is equivalent to $\{$f(x_1),f(x_2),...,f(x_n)$\}$).

Figure 5.22 shows the HorseIR program for the example query in Figure 2.5 with a scalar UDF. The HorseIR code consists of a module with methods: the query is

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

Figure 5.22 – HorseIR code with a new method for the UDF in Figure 2.5

translated to the main method, and the UDF is translated to the method `calcRevenueChangeScalar` which takes two arrays of type float as input and returns the resulting product. This integrated HorseIR code, which combines the SQL and the UDF part, can now be further optimized. We will discuss the code optimization for HorseIR generated from HorseUDF in Section 7.4.

Chapter 6
Back-end: Execution on HorseIR

Since HorseIR is an intermediate representation for database queries and array languages, it needs to be executed directly or further compiled to low-level efficient code. Figure 6.1 presents the overview of our design: HorsePower supports an interpreter which executes HorseIR directly (Section 6.1); it provides a compiler which compiles HorseIR to annotated C code, and the C code is then compiled using C compilers to generate binary code for the CPU and GPU (Section 6.2); and execution of the compiled binary code is further supported by a prepared library having a set of pre-compiled high-performance built-in functions that can be integrated into the interpreter and the compiled binary code (Section 6.3). Runtime support for managing data is also necessary as the code needs to run over the data stored in memory (Section 6.4).

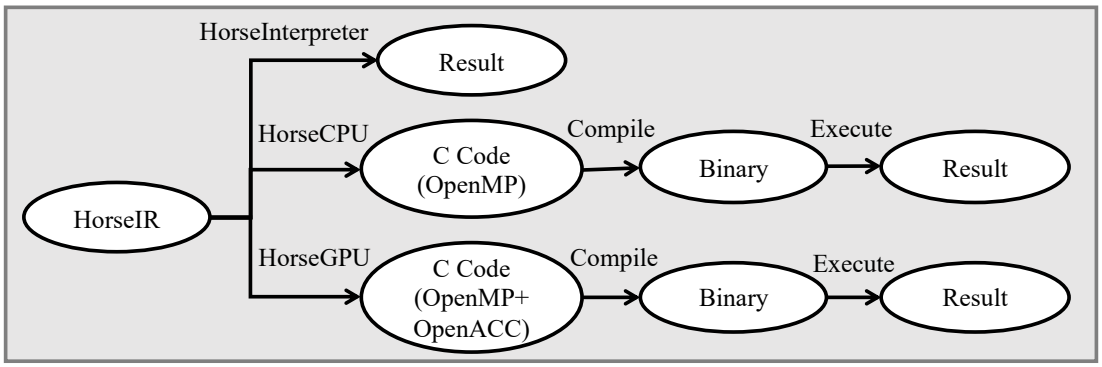


Figure 6.1 – Overview of HorseIR back-ends

6.1 HorseInterpreter

HorseInterpreter is an interpreter-based engine for executing HorseIR. We have mainly used it validate the correctness of the HorseIR code generated by the HorsePower front-end. With an interpreter-based design, HorseIR can be executed directly after parsing without going through the code compilation for generating target code. As an array-based language, a variable in HorseIR represents a collection of data, such as vector and list, and a built-in function can be overloaded with different types and shapes of input parameters. Therefore, the interpreter is designed to keep track of these variables and offer a pre-compiled library to support these built-in functions. The implementation strategy for these library functions can be found in Section 6.3.

The naive interpretation mode may introduce intermediate results at multiple points during the execution. This may significantly influence performance because program optimizations could enable the merge of two or more operations in HorseIR, avoiding the need to explicitly generate and save an intermediate result. For example, assume a vector of prices is multiplied with a vector of discounts in an element-wise manner before total prices are calculated as follows:

```
1 ...
2 t0:f64 = @mul(price, discount);  // get new prices after discounts
3 t1:f64 = @sum(t0);               // get total prices after discounts
4 ...
```

Let `price` $= < p_0, p_1, ..., p_{n-1} >$, and `discount` $= < d_0, d_1, .., d_{n-1} >$ be two vectors of length n. Then, the variable `t0`, which is an intermediate vector, contains $< p_0 * d_0, p_1 * d_1, ..., p_{n-1} * d_{n-1} >$ after element-wise arithmetic multiplication of the two input vectors `price` and `discount`. The variable `t0` is later reduced in the function `sum` to return the total price after discounts, saving the result in `t1`. When the number of rows is small, the cost of having intermediate results is negligible even if the variable `t0` is only used once. However, it could become a severe performance bottleneck with large vectors. Ideally, this overhead can be reduced by fusion-based optimizations, which we will introduce in Chapter 7. In this example, the two statements can be

simplified into a math equation:

$$t1 = \sum_{i=0}^{n-1} p_i * d_i$$

which could be written in a single loop for accumulating the result after multiplication. Such optimizations are not performed in the interpreter mode. Nevertheless, despite the naive approach, HorseInterpreter presents a reliable system that outputs the correct result of HorseIR programs.

It should be noted that HorseInterpreter at the current stage lacks sophisticated techniques for an interpreter, such as automatic garbage collection and just-in-time compilation support. This is due to the fact that HorseInterpreter was originally designed as a prototype system without performance consideration as we handle optimizations in our compiler, presented in Section 6.2. However, there is still potential future research for the interpreter, which executes code directly without the penalty of compiling the generated code to binary code as a compiler does, which can improve the response time of ad-hoc queries.

6.2 HorseIR Compiler

In order to improve the performance of HorseIR, we designed and implemented a compiler to generate efficient target code for both the CPU and the GPU. The optimizations integrated into our HorseIR compiler are specifically tailored for HorseIR code generated from execution plans of database queries and MATLAB programs.

Figure 6.2 shows an overview of the HorseIR compiler. We follow a standard design, which first parses the HorseIR source code and compiles it to an abstract syntax tree (AST). A clean grammar for HorseIR can be found in Appendix A.1. Then, a weeder is implemented to validate constant values. For example, the date type requires the range of day to obey the rules defined in Appendix A.2, such as that it is invalid to have a month which has more than 31 days. Next, after constructing symbol tables and propagating type and shape information, the annotated AST nodes

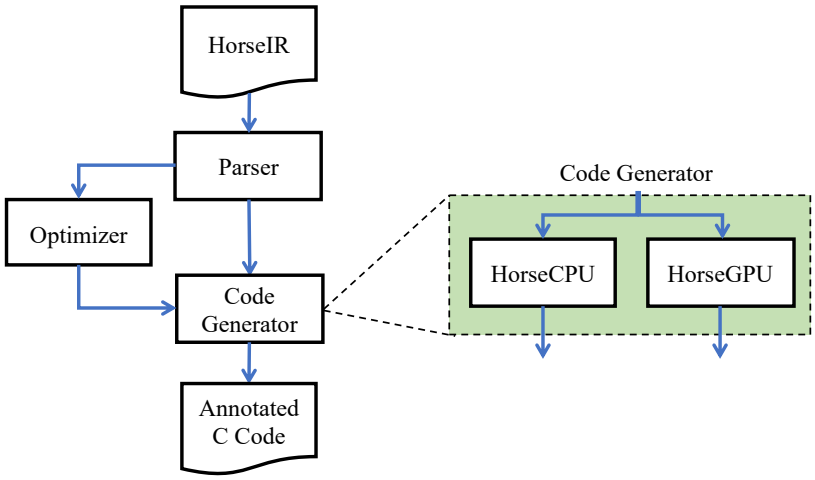


Figure 6.2 – Overview of HorseCompiler

have sufficient information to generate target code. Such directly generated code is called *naive* code since it has little optimizations. Alternatively, the compiler can pass these AST nodes to the next stages of optimizations and generate *optimized* code. The HorseIR optimizer is designed for collecting precise program information, such as type, shape, and data dependency, in order to generate efficient target code. Finally, we have two code generators: *HorseCPU* for generating CPU code only, and *HorseGPU* for generating hybrid code containing both CPU and GPU code. Both generators create annotated C code with the support of OpenMP and OpenACC separately for parallel computing.

The design of HorseIR enables simple parallelization and exposes information for optimization generally, the HorseIR compiler generates target code with invocations to pre-defined functions in the built-in function library. However, this is not always the approach taken. In particular, parallelizing a single function is often inefficient because an implied synchronization barrier introduced after each operation is expensive. Thus, it is often beneficial to merge two or more functions by generating code on which loop fusion based optimizations can be applied. Thus, at compile time, depending on the available type and shape information, the compiler decides whether to use the pre-compiled function code or it generates ad-hoc code that can then be optimized via fusion to generate specialized C code as described in Chapter 7.

6.2.1 HorseCPU

HorseCPU is a code generator for generating CPU code only. It has two versions of target code: the *naive code* without any optimizations, and the *optimized code*. The naive code has almost the same performance as the HorseInterpreter as they employ the same strategy by invoking pre-compiled built-in functions, and thus their performance highly depends on the built-in function library. Furthermore, they both can encounter the performance problem of having intermediate results generated between function invocations. They mainly differ in compilation time, since HorseInterpreter can execute HorseIR code once it is parsed, while the naive code needs to be compiled to binary before being executed. Therefore, we explore two general optimization strategies for HorseIR to generate more efficient code.

- **Fusion-based optimizations.** This technique aims at fusing HorseIR statements for array operations in order to generate compact code with fewer loops and intermediate results. We implement both automatic loop fusion and pattern-based fusion to merge various array-based operations as described in Section 7.3.

- **Source-level optimizations.** Since HorseIR can be generated from various sources, it is inevitable that HorseIR may contain "dead" code that is never used in getting the final result. We implement *backward slicing* [84] to remove such dead code in order to compact the code and reduce memory footprint. We also consider *method inlining* [26] to merge methods, giving a larger codebase for optimizations within methods. This is helpful when HorseIR methods are generated from query and UDF sources that need such holistic optimization. The details of source-level optimizations can be found in Section 7.1.

The generated optimized code is based on C code with OpenMP parallel directives, which enables simple parallelization strategies based on additional directives on the top of C code. Example code looks as follows:

```
#pragma omp parallel for simd reduction(+:total)
    for(int i=0; i<n; i++){
        total += x[i];
    }
```

This code computes the total value of the vector `x` and stores the value in the reduction variable `total`. The first line, which contains several keywords, annotates the C code and tells compiler to generate parallel code. The keywords `#pragma omp parallel for` presents a parallel loop construct; the keyword `simd` enables the SIMD code generation if applicable; and the keyword `reduction` defines a reduction variable `total` with the operator `+` (i.e., sum). The for-loop after the first line is influenced by these parallel directives. The C compiler will generate parallel binary code for the loop automatically. By using C macros, we can define a parallel for-loop as follows:

```
#define STRINGIFY(x) #x
#define DOP(n, x, ...) {int i2=n; \
    _Pragma(STRINGIFY(omp parallel for simd __VA_ARGS__)) \
    for(int i=0;i<i2;i++) x;}
```

The macro `DOP` has two parts: the first line for OpenMP directives and the for-loop. It uses another macro `STRINGIFY` to generate the string values for OpenMP directives inside the macro `_Pragma`. An arbitrary number of arguments can be given with the built-in keyword `__VA_ARGS__`. The for-loop is simple with a loop body stored in the parameter `x`. Thus, the original C code can be written as:

```
DOP(n, total+=x[i], reduction(+:total));
```

Other than having these macros in the code generation, we use them to implement our built-in function library, greatly reducing the amount of effort in writing parallel code, and improving the quality of the code.

In the optimized code generation, we employ a hybrid strategy combining the built-in function library and the generated C code with the purpose of lowering compilation time: (1) when fusion is possible, multiple operators are fused to generate fused C code; otherwise (2) each operator is compiled to an invocation to its built-in

function in the library. The details of parallel code generation strategies can be found in Section 7.3.

6.2.2 HorseGPU

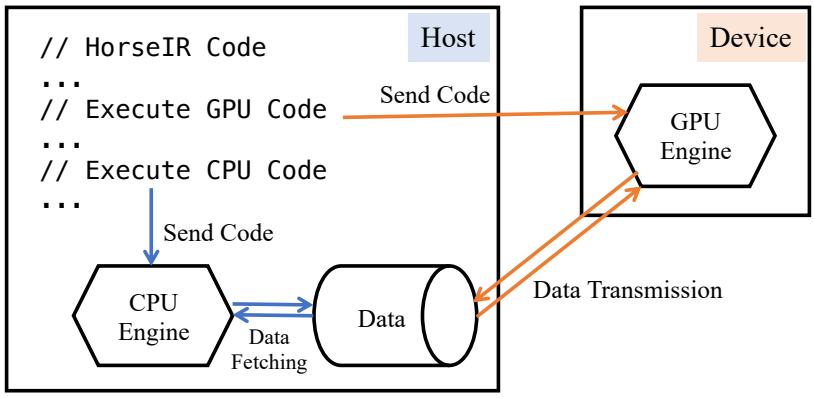


Figure 6.3 – Overview of HorseIR working with GPUs

HorseGPU is a code generator for generating hybrid CPU and GPU code. As depicted in Figure 6.3, HorseGPU needs to handle the CPU code on the host side and the GPU code on the device side. When executing code on the host (CPU), it can fetch data from the memory. However, it is required to transmit data from the main memory to GPU memory when the code needs to be executed remotely. Therefore, the performance of the overall execution needs to consider the CPU time, the GPU time (kernel execution time on the device), and the data transmission time.

The optimizer determines a set of HorseIR statements, which are selected to be sent to the GPU. They are called the *selected code*. Executing the selected code on the CPU is estimated to be more expensive than the combined time of the data transmission and the kernel execution on the GPU. For the non-selected code it is, in contrast, estimated to be cheaper and we should avoid deploying such execution on the GPU. However, it is hard to precisely estimate when it is beneficial to use the GPU. For our current system, we identify a set of expensive built-in functions that are much slower on the CPU than the GPU, for example, math functions `exp` and

`log`. Thus, the cost of data movement becomes acceptable with much faster kernel execution on the GPU.

We also look into the fusion-based optimizations developed for HorseCPU (Section 6.2.1) even though they are mainly for the CPU architecture. The goal of fusion is to reduce the intermediate results, and this also fits the idea of optimizations for HorseGPU, i.e., reducing data movement between CPU and GPU. We introduce a new strategy in the optimizer that leverages the fusion techniques and generates GPU code for the fused statements when expensive built-in functions are involved.

Like HorseCPU, HorseGPU also has two versions of target code: the *naive code* without any optimizations, and the *optimized code*. The optimized code is mixed with OpenMP for CPU code and OpenACC for GPU code. We provide a GPU-based library, which contains a parallel GPU implementation for a subset of built-in functions. The code in Figure 6.4 is an example of OpenACC which copies the vector `x_ptr` with the length `n` and returns the vector `z_ptr`. The core computation of the code is executed on the GPU after the data copy finishes.

```
1 #pragma acc data copyin(x_ptr[0:n]) copyout(z_ptr[0:n])
2 {
3 #pragma acc parallel loop
4     for(int i=0; i<n; i++){
5         z_ptr[i] = exp(x_ptr[i]);
6     }
7 }
```

Figure 6.4 – Example of OpenACC code

As can be seen, OpenACC is similar to OpenMP in providing well-formed parallel constructs. It also supports OpenACC-specific directives: the keyword `acc` defines OpenACC directives; and the keyword `copyin` copies a specified size of the data from the host (CPU) to the device (GPU); and the keyword `copyout` copies a specified size of the data from the device (GPU) to the host (CPU). We extend our implementation to support OpenACC code by explicitly adding the input and output of the data in each block of computation. This is particularly convenient for code generation from HorseIR, as it has many built-in functions.

6.3 High-performance Built-in Function Library

HorseIR provides a rich family of built-in functions, as we introduced in Section 4.4 and in Section 5.1. In this section, we present the implementation details of some of these functions. HorseIR employs a single-function-multiple-implementation strategy to embrace the various kinds of data from database systems. One built-in function may have one or more implementations that are specialized to the correct base type, or the size, or shape of the input data. We use OpenMP and OpenACC to implement parallel C code with SIMD vectorization enabled in the library, and the efficient parallel code generated by fusing element-wise built-in functions, as described in Chapter 7. One built-in function may thus have both CPU and GPU implementations. At compile time, our optimizer decides which implementation should be used. The list of built-in functions in the library can be found in Appendix A.3.

6.3.1 Basic Built-in Functions

A large subset of built-in functions having parallel implementations are influenced by array programming. Their parallel implementation strategies lie in the following categories:

Embarrassingly parallel: We have a large set of functions which are designed for embarrassingly parallel tasks, including element-wise functions, the pattern-matching function `like`, and the searching function `member`. There is no dependency or communication between parallel tasks. For instance, both unary and binary element-wise operations maintain one-to-one correspondence without data dependence. Thus, the HorseIR library implements a parallelized version for each of them supporting different input types. The functions `like`, `member`, and `index` are special because they have two inputs, one for matching or searching, and another one for iterating. Still, they can be implemented with an embarrassingly parallel strategy.

Reduction functions: The reduction result can be obtained by using the OpenMP reduction clause. For example, the function `sum` for returning the total value

can be implemented with `reduction(+:t)` where the variable `t` is the reduction variable storing the final result. A code example for returning the total value of an input vector has been given in Section 6.2.1.

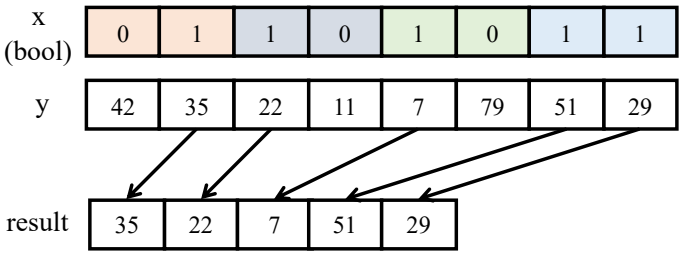


Figure 6.5 – Example of the compress function

Boolean selection function: We show the parallelization of our boolean selection function `compress` along an example. As can be seen in Figure 6.5, the function `compress` takes two same-length vectors: a boolean vector `x`, and a vector `y`, and it returns the values from `y` at the indices for which the corresponding indices in `x` are true (i.e., 1). Assuming there are four threads, we first partition the vector `x` into four sections (shown with different colours in the figure). We first have to know the size of the output vector. For that, we compute (in parallel) the number of true values in each section and get `{1,1,1,2}`. From there, we can calculate the prefix sum starting from 0 and get `{0,1,2,3}`. These numbers are the beginning positions of partitions for writing the final results. Therefore, we can achieve parallel processing by partitioning vectors and computing prefix sum.

6.3.2 Important Database-related Functions

For the database operations such as `join_index` and `group`, the concept of one-strategy-fits-all is not always appropriate, as the implementation of database operations is greatly influenced by input data, considering the size, type, and range of the data. Each of possible implementation has its own limitations so that the trade-off

between these strategies needs to be carefully considered. We present a couple of strategies used in implementing our library as follows:

Join: The `join_index` operation takes two input parameters (left and right) and returns a list containing two integer arrays as output as was described in Section 5.1.1.3 and illustrated in Figure 5.5. Each parameter can consist of one or multiple arrays and both arguments should agree on the number of arrays. The join operation searches for values that can be found in both input parameters. It then returns two equal-length integer arrays indicating the rows/indices where the two inputs have the same value.

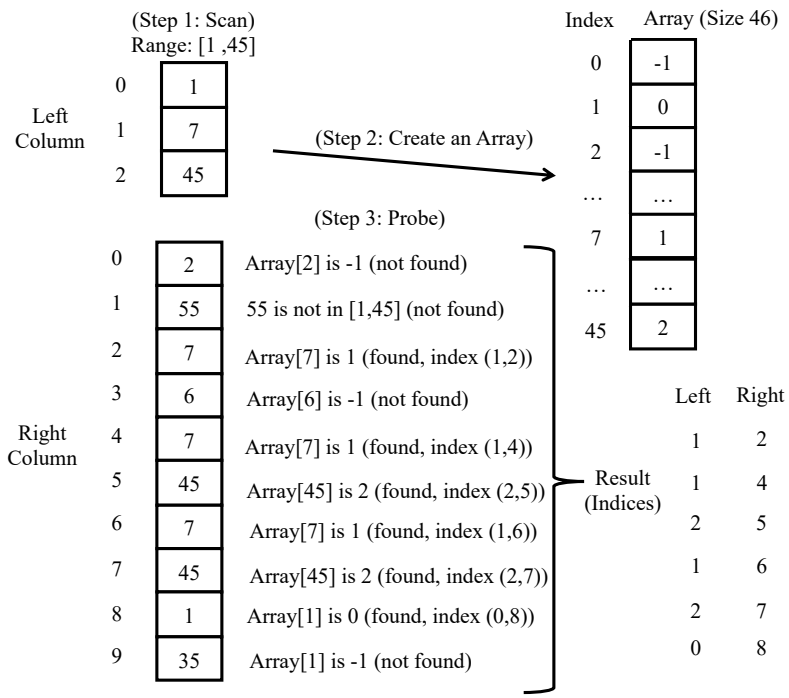


Figure 6.6 – Design of array-lookup join

- Figure 6.6 presents an example execution using the *array-lookup* join implementation, designed for equal joins when the left argument has one integer array which contains unique data. First, we scan the left column

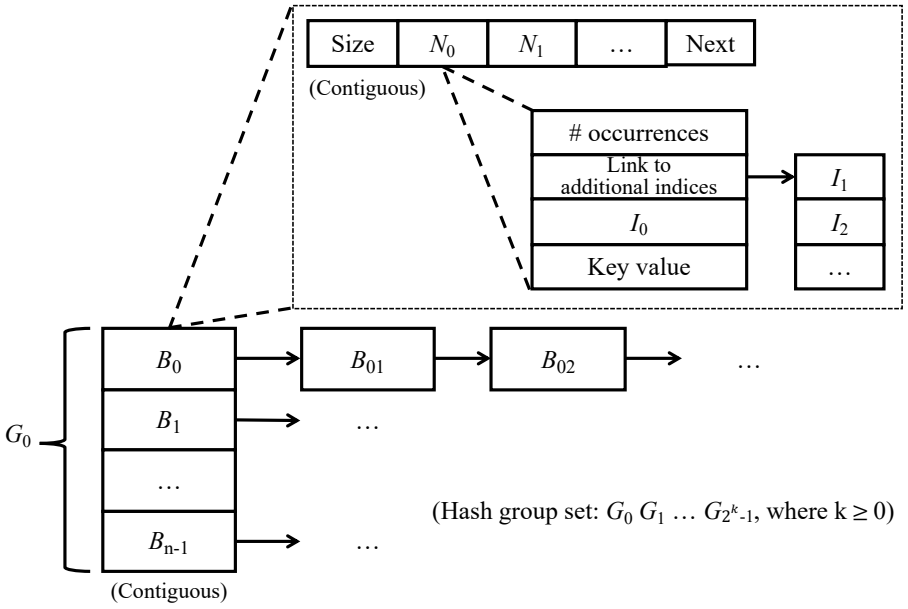


Figure 6.7 – Design of radix-based hash join

to get the range of the array, in the example from 1 to 45 inclusive. Furthermore, we check whether the left column follows a strict ascending or descending order (no equal neighbours). If not, we sort the data and check again. In the example, the input left column is already sorted. Next, we create an *index array* to cover the value range (initialized with -1) and assign each index the position in the left input column that has this index as its value. The corresponding indices from the left column are filled in the array. After that, we probe each position in the right column to see if its value is in the range, and if yes check whether the corresponding position in the index array is non-negative. If this is the case the position indicated in the index array and the current position in the right column are added, and the value found in the intermediate integer as output values in the result array as shown in the right bottom of Figure 6.6.

- Figure 6.7 depicts the design of the main data structure used in a *radix-based hash join* implementation. The radix-based hash join algorithm [15] is an efficient strategy for minimizing cache misses. The data structure is

a special hash table. Assume a column with integer values, the hash table allows you to quickly find the indices of the column that hold a given value. The hash table consists of a set of hash groups (G) and each of them is a smaller hash table. Given an integer value, we can determine to which group it belongs by looking at its lower bits. For example, when the number of hash groups is 256, we only need to look at the lower 8 bits of an integer. Each hash group consists of an array of contiguous buckets, and each of them points to the next bucket if available. In a bucket, there is another contiguous space storing "Size" for the number of used hash nodes, and "Next" for connecting to the next bucket. A hash node keeps track of all the indices of one unique integer value. As such, it stores the number of times a value occurs in the input column, the first index in the input column that holds the value, the link to a contiguous array for storing all other indices, and the actual value of the integer. If a value occurs only once, the number of occurrences is 1 and the pointer to the other indices is set to NULL. As can be seen, the design of the hash table avoids pointers, but uses contiguous arrays that can improve the performance of hash tables by reducing cache misses during hash probing. In the actual join operation, one column is parsed and the hash table is generated. Then the other columns are parsed and for each row, the corresponding indices are found in the hash table.

- *Mergeable hashing* is for input arrays which can be merged into a single array. This is possible in some cases. For example, two arrays with the type integer (32 bits) can be transformed into an array with the type long (64 bits). Based on the data range of the new array, we then choose either array-lookup or radix-based hashing for the actual join.

- *Hybrid hashing* is employed when equal and non-equal joins are both found in multiple-column joins. We first perform the equal joins and then fetch indices which are later used in the comparison for the other columns including non-equal joins. This assumes that the equal join is usually more

efficient than non-equal joins.

Sorting: Given an input array, sorting returns an output array $[a_0,a_1,...,a_{n-1}]$ of the same length containing the same values as the input array but the values are sorted in ascending order ($a_i \leqslant a_{i+1}$) or descending order ($a_i \geqslant a_{i+1}$).

- *Radix sort* [91] is a fast sorting algorithm for integers. It scans a set of bits of integers from lower bits to higher bits, and maintains enough buckets to store these integers based on the currently scanned bits. After the scan for all bits completes, the array of integers is sorted accordingly. The advantage of radix sort is that the whole process avoids data comparison, although the need to make a copy of the whole array is inevitable after each round of scan.
- *Quicksort* [39] is a fast sorting algorithm for all types of data. It is implemented to support non-integer input data, such as floating numbers and compound data (i.e., multiple columns). Furthermore, it provides a general interface for supporting multiple-column sorting with various orders. For instance, the first column is ascending while the second column is descending.
- *Parallel merge sort* is a variation of the basic merge sort by utilizing multicores. The merge sort recursively divides the data into two parts, and aggregates the two parts in proper order. We employ OpenMP parallel constructs to implement parallel merge sort: parallel code for dividing the data and synchronization for aggregating the parts. In addition, we introduce radix sort into the merge sort: when the size of the current part is less or equal than a threshold value, the radix sort operates on the entire part. It also can work with other sorting algorithms, such as quicksort.

Groupby: The `group` operation takes one or multiple arrays as input and groups the indices having the same value as described in Section 5.1.1.5 producing an dictionary structure as illustrated in Figure 5.8. We currently implement sort-based grouping that first performs sorting before scanning adjacent neighbours.

This is inexpensive when the size of input data is small. However, it could be a performance bottleneck when the size grows larger. In particular, sorting on multiple arrays is slow since all columns have to be sorted with poor cache efficiency. Therefore, we also implement a hash-based algorithm for grouping when the input data is large. In addition, we support efficient grouping implementations when the following specific kinds of input data are identified.

- If the input data is already sorted, the step of sorting can be omitted. A scan on the input data is used to determine if the input data is ordered. If ordered, we employ a parallel implementation: we first divide the data into multiple segments based on the number of threads, and examine adjacent items within each segment to collect the group information. Next, we continue examining items across segments, i.e., checking the data at the two ends of a segment, and update the group information. In the end, we write out the final groups

- *Mergeable data* can be used for the grouping on multiple arrays, if these arrays can be merged to a single array before the actual grouping occurs. A typical example is the merge of two `int` arrays into a `long` array because the type `int` has 32 bits and the type `long` has 64 bits. The cost of merging arrays is non-negligible, but the overall performance can be improved because of better cache efficiency in later sorting.

- *Special data* has an integer type which represents a short range, such as `char` (8 bits), `small` (8 bits), and `short` (16 bits). Instead of sorting the whole data, we can maintain a table, which covers the range of data types, that records all the indices with same value. With additional space cost, we can achieve better performance for such integers without actual sorting.

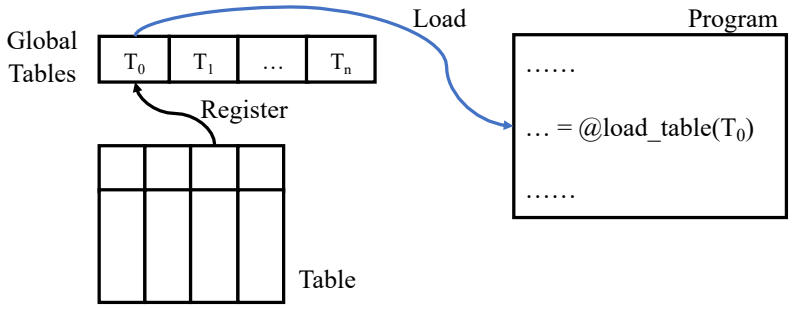


Figure 6.8 – Global table registration and fetching

6.4 Data Management

As a system designed for handling database query processing, HorsePower provides facilities to manage data stored in memory and feed the data during program execution. It treats database tables, the most fundamental element in database systems, as a `table` type. It needs to implement functions to register, load, and maintain tables. Figure 6.8 shows that a raw table file is first loaded into main memory, then registered into a global registry, and later loaded to the execution environment. If a load request cannot find the table in the registry, an error is raised.

It should be noted that a temporary table can be generated during the execution by using the built-in function `table`. This commonly exists in standard database queries when a table needs to be returned as the result of a query. Since such tables are considered as temporary tables, they are not registered, and the function `load_table` is unable to identify them. An extra step of registration is needed if a temporary table is to be recognized as a global table.

With the main design purpose of supporting data analytics, HorsePower is unable to modify the content of an existing table, but it is allowed to create a new table. This is different from the table conversion between tables with and without keys, as described in Section 4.3.2, that only modifies table structure but leaves table content unchanged. At the stage of data initialization before executing HorseIR programs, we load data and create tables with primary and foreign keys as needed.

Chapter 7
Optimizations

HorsePower provides an optimizer that compiles HorseIR to efficient C code. It is critical to have the optimizer since the design of HorseIR with three-address code introduces additional intermediate results that can lead to poor performance in a naive interpreter or compiler. The optimizer supports conventional static program analysis and collects precise type and shape information for subsequent fusion-based optimizations to generate efficient target code with fewer intermediate results. Moreover, the optimizer supports cross-method optimizations for the HorseIR code generated from database queries with UDFs written in MATLAB.

An overview of the optimizer workflow can be found in Figure 7.1. We first generate HorseIR from optimized query execution plans, MATLAB programs, or both as introduced in Chapter 5. Next, method inlining merges methods into a larger codebase to explore further optimizations, and program slicing removes possible unused code. We describe the details of these steps in Section 7.1. After that, type and shape analysis is performed to process the HorseIR program and compute the type and shape information at each expression. Types and 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 based on a data-dependence graph. The details of this process are described in Section 7.2. Lastly, in Section 7.3, we employ code generation optimizations by leveraging the set of fusible sections to generate fused target C code. This is supplemented by using

patterns to exploit additional optimization opportunities that are frequently present in SQL queries.

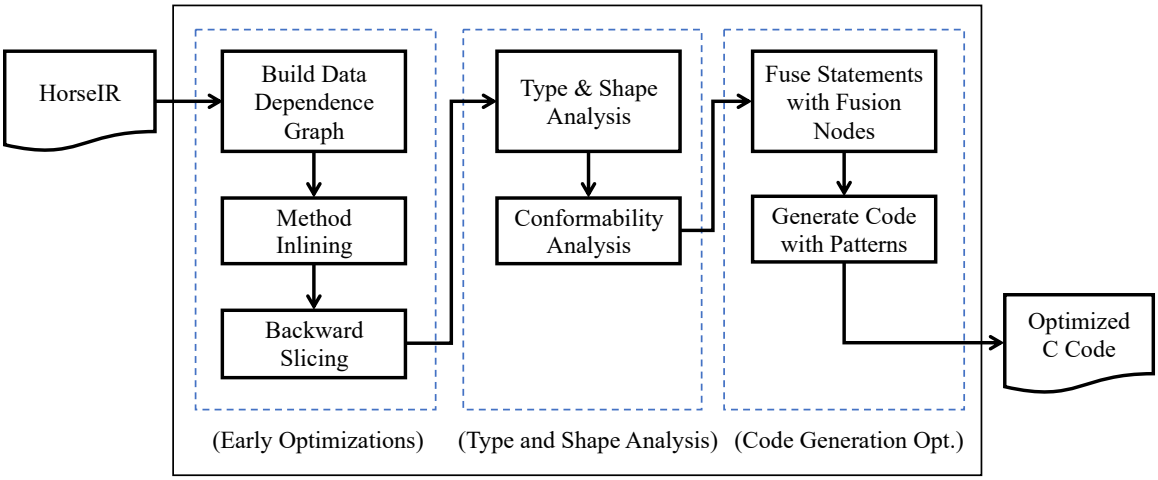


Figure 7.1 – Analysis and code generation overview.

7.1 Early Optimizations

The compiler for HorseIR follows the conventional compiler design for optimizing HorseIR code, such as building data dependence graphs and providing dataflow analysis support. Thus, a variable or method in a statement knows its all definitions in other statements in a use-definition chain (UD Chain), and a variable or method definition in a statement knows its all uses in other statements in a definition-use chain (DU Chain). Both chains can be constructed by using reaching definitions [62] with the input of data dependency graphs.

As an intermediate language, HorseIR can represent programs from various source languages, such as database queries and MATLAB. Even though the translators from these languages can generate optimized HorseIR code individually, it is still possible to introduce dead code or redundant code when combining HorseIR code from, for example, database queries and MATLAB programs, as introduced in HorseUDF in Section 5.3. This first optimization phase addresses this by using method inlining for

merging methods, and backward slicing for removing code that cannot contribute to the final result.

Method inlining. When a method is inlined, the call of a method is replaced by the corresponding code segments that constitute the method being called. Thus, the caller method has now a larger codebase for exploring more optimization opportunities so that the overall execution time can be improved [26], such as by facilitating the elimination of unused computations or code fusion across methods. 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. By having all the code in a method, this can be detected through backward slicing to avoid the computation of the unused column in the table UDF. Moreover, method inlining bridges methods, which are generated from SQL queries and UDFs, to explore more fusion opportunities.

While performing inlining, to respect the pass-by-value convention for parameter passing, a copy of an object used as a 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.

We provide more details about this cross-optimization in Section 7.4, after we cover the individual analysis and optimizations in Section 7.2 and Section 7.3.

Backward slicing. Program slicing [84] is a common compiler technique for finding all related statements at a specific program point. Based on the use-definition information, a statement can trace back all previous statements with the flow to the current statement. Backward slicing is implemented to scan from the final result to identify the code which needs to be retained, and eliminate all other code, as it is effectively dead code.

There are a couple of scenarios in which this technique can be particularly effective. One scenario is when HorseIR code is generated from a front-end which lacks optimizations for removing such dead code. For example, in Section 5.1.2, HorseIR code generated from execution plans may contain dead code for materializing columns that are never used later on. Thus, the code generating these columns can be removed by using backward slicing. Another scenario is when methods are inlined but the caller does not require all the functionality the callee provides. Parts of the inlined code could then be eliminated as dead code.

7.2 Type and Shape Analysis

Type and shape information is important for subsequent code optimizations to generate efficient code. Our optimizer supports type propagation (Section 7.2.2) and shape propagation (Section 7.2.3) to collect precise type and shape information. HorseIR provides around 100 built-in functions, and each built-in function has its own type and shape rules. A complete list of built-in functions and the description of these functions can be found in Appendix A.3. We will present a couple of typical built-in functions to demonstrate how type and shape propagation works with these pre-defined rules. Type and shape rules for more built-in functions are described in our online documentation [4].

7.2.1 A Motivating Example

Figure 7.2a shows a simplified version of Query 6 of the TPC-H benchmark [85] computing the change in total revenue given prices and discounts from the table `lineitem` for all those items where the discount is at least 0.05 as we had already seen in Figure 2.2. A basic translation of this query into a HorseIR program (prior to performing any optimizations) is shown in Figure 7.2b. In the HorseIR program, the function `main` defines the entry of the program and returns a table as the final result.

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

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

```
module ExampleQuery{
  def main(): table{
    // load table
    t0:table = @load_table(`lineitem:sym);
    // load two columns
    t1:f64 = check_cast(@column_value(t0, `l_extendedprice:sym), f64);
    t2:f64 = check_cast(@column_value(t0, `l_discount:sym), f64);
    // compute revenue change
    t3:bool = @geq(t2, 0.05);      // S9
    t4:f64  = @compress(t3, t1);   // S10
    t5:f64  = @compress(t3, t2);   // S11
    t6:f64  = @mul(t4, t5);        // S12
    t7:f64  = @sum(t6);            // S13
    t8:sym  = `RevenueChange:sym;
    t9:list<f64> = @list(t7);
    t10:table = @table(t8, t9);
    return t10;
  }
}
```

(b) HorseIR code for (a)

Figure 7.2 – Example query and its HorseIR program

```
1 ...
2 revenue = 0;
3 length_t2 = ...;
4 for(i = 0; i < length_t2; i++){
5     if(t2[i] >= 0.05)
6         revenue += t1[i] * t2[i];
7 }
8 ...
```

Figure 7.3 – Optimized C code for the IR code in Figure 7.2b

It first obtains the reference to the table `lineitem`[1] and then uses it to obtain the references to the data sets of the columns `l_extendedprice` and `l_discount` (both being vectors) in variables `t1` and `t2` respectively. Then, the predicate is evaluated by invoking the built-in function `@geq`. This function returns a boolean vector of the same length as `t2` with a `true` value for each row where the corresponding `t2` row has a value of at least 0.05, and a `false` otherwise. Next, the function `@compress` in lines 12 and 13 takes this boolean vector and `t1` respective `t2` as input, and returns all rows from `t1` respective `t3` where the corresponding values in the boolean vector are `true`. The length of the resulting vectors `t4` and `t5` is equal to the number of true values in the boolean vector `t3`. Finally, the multiplication function in the `SELECT` clause is executed on the two vectors in lines 14, and the summation in line 15. In the end, in lines 16 to 19, a new table, with a single column named `RevenueChange` and a single row for the total revenue change, is created and returned.

As can be seen, such an approach generates a fair number of intermediate results. In particular, `t3` to `t6` are all intermediate vectors that are materialized. If lines 11 to 15 are translated to lower-level code independently, each of them generates its own `for` loop over the corresponding arrays. However, array-based optimization techniques, including loop fusion, and some pattern-based optimizations developed specifically for the operator sequences found in SQL statements, allow the HorseIR compiler to fuse these loops to just one loop to avoid materializing these intermediate vectors. The resulting sequential C code after such optimizations is similar to the

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

one depicted in Figure 7.3. The various optimization techniques will be discussed in-depth in the rest of this chapter. Although the source code in Figure 7.3 does not convey it explicitly, behind the scenes, HorseIR uses OpenMP to compile them into a parallel implementation, as outlined in Section 6.2.

7.2.2 Type Propagation

HorseIR provides explicit type but implicit shape information. For instance, a vector integer type `i32` indicates a vector with the type integer and unknown length; two variables having different types may hold the same length. The existence of the wild-card type requires a proper type propagation to find out its specific type information at compile-time.

Abstraction for types. The abstraction of a HorseIR type needs to keep track of a vector type with a main type, or a list-based type with a main type and cell types. Therefore, we implement a type node for preserving all type information as illustrated in Figure 7.4. A type node has three fields: the main type, a pointer to a type node as a cell type, and a pointer to a type node as an adjacent type. It is able to represent an integer when the two pointers are set to null, and it is also flexible to represent lists with the pointer fields.

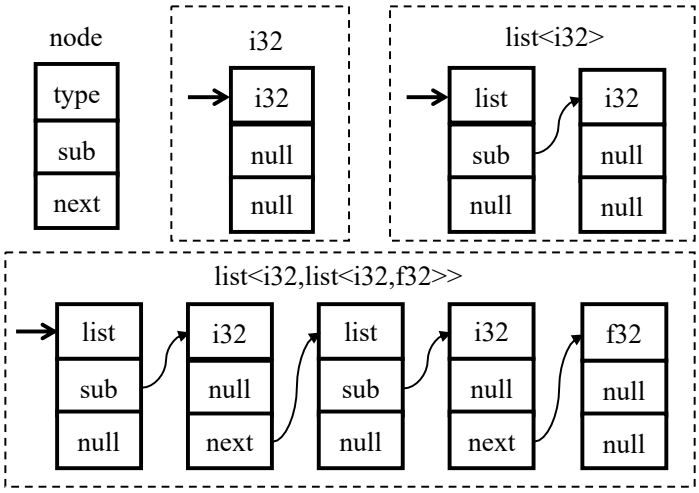


Figure 7.4 – Type nodes for representing type information

Initial types. Since HorseIR code reads data from tables in main memory and each table has a set of well-defined columns with specific types, the optimizer gets the initial types from these columns. For example, the below code loads a column `l_discount` and checks if it is a column with the type `f64`.

```
1 t2:? = check_cast(@column_value(t0, `l_discount:sym), f64);
```

Thus, although the type of the assigned variable `t2` is a wild-card (`?`) it can be specialized with a type `f64`.

Type propagation. The abstract information of type information is propagated through statements and methods. HorseIR provides pre-defined type and shape rules for all built-in functions. Since HorseIR supports dozens of types, we use the simple letters for types introduced in Table 4.1 and Table 4.2 to describe type rules. For example, the letter `B` denotes the boolean type.

As an example, Figure 7.5 shows the type rules for boolean binary built-in functions, such as equal (`eq`) and not equal (`neq`). If the input is any two real types (`B/J/H/I/L/F/E`) or otherwise both inputs have the same specific types (`C/Q/S/M/D/Z/U/W/T`) the result type is Boolean. An empty slot in the figure represents an unhandled case, and a type error should be raised.

7.2.3 Shape Propagation

The abstraction of a HorseIR shape needs to understand the symbolic value of current sizes. A shape may maintain different kinds of status: *constant*, *non-constant*, and *unknown*. The HorseIR optimizer implements a symbolic system which can reduce the number of kinds by merging non-constant and unknown shapes, because they can be assigned with a unique symbolic value when an unknown shape occurs. However, it depends on the precision of the symbolic system to provide shape information for subsequent optimizations, such as fusion-based optimizations for merging loops with the same iteration when generating C code from array-based built-in functions. The below code example from Figure 7.2b shows how shape information can affect the final code generation.

y \ x	B	J	H	I	L	F	E	C	Q	S	M	D	Z	U	W	T	X	G	N	Y	A	K
B	B	B	B	B	B	B	B															
J	B	B	B	B	B	B	B															
H	B	B	B	B	B	B	B															
I	B	B	B	B	B	B	B															
L	B	B	B	B	B	B	B															
F	B	B	B	B	B	B	B															
E	B	B	B	B	B	B	B															
C								B														
Q									B													
S										B												
M											B											
D												B										
Z													B									
U														B								
W															B							
T																B						
X																						
G																						
N																						
Y																						
A																						
K																						

Figure 7.5 – Type rules for boolean binary functions with two parameters x and y

```
1  ...
2  t1:f64 = check_cast(load_column(t0, `l_extendedprice:sym), f64);
3  // array size: d0 (l_extendedprice)
4  t2:f64 = check_cast(load_column(t0, `l_discount:sym), f64);
5  // array size: d0 (l_discount)
6  ...
7  t3:bool = @geq(t2, 0.05:f64); // array size: d0
8  t4:f64  = @compress(t3, t1);  // array size: d1
9  t5:f64  = @compress(t3, t2);  // array size: d1 or d2
10 t6:f64  = @mul(t4, t5);       // array size: d1 or d3
11 t7:f64  = @sum(t6);           // array size: 1
12 ...
```

As can be seen, a symbolic number `d0` is assigned to the number of rows in the table `lineitem`. In order to keep it simple, a growing numeric id is adopted to represent new and unique symbolic values. Two columns `l_extendedprice` and `l_discount` are first loaded from a table `lineitem`. Therefore, the two columns have the same size `d0`. Next, the column `l_discount` is input to the greater-than-or-equal-to function `geq` to return a boolean vector of the same size indicating whether each discount is greater than 0.05. Then, the result vector is used to select qualified discounts with the function `compress`. The size of `t4` is determined by the number of true elements in the boolean vector `t3`. However, the actual size is only known at runtime. Thus, a new symbolic size `d1` is assigned to the vector `t4`. For the size of `t5`, there are two cases: (1) `d2`, if the symbolic system naively assigns the next available symbolic number; or (2) `d1`, if the optimizer understands the variable `t3` which is used in the previous statement. It should be noted that the size `d1` is assumed always to be not equal to `d2`. Furthermore, the size of `t6` after multiplication may be either `d3`, a new size, in case of (1); or `d1` in case of (2). Finally, it returns the size 1 after computing the total value of `t6` with the function `sum`. In case (1), the optimizer is unable to understand that the two statements for boolean selection share the same boolean vector, whereas in case (2), the statements in lines 7-11 can be fused into a single loop as they share the same loop iteration as shown in Figure 7.6. Hence, we introduce

```
1 // ... load columns t1, t2
2 t7 = 0;
3 for(int i=0; i<n; i++){
4     if(t2[i] >= 0.05){
5         t7 += t1[i] * t2[i];
6     }
7 }
8 // ... return t7
```

Figure 7.6 – Generated fused C code

a systematic shape analysis by defining precise shapes, setting up propagation rules, and finding statements that can be fused.

7.2.3.1 Shape Analysis

Shape information is key to identifying 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.

Shape Abstraction

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

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 7.1.

Table 7.1 – 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 which is 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 the boolean mask `a`. The code generator uses this boolean selection shape for further fusion and avoids storing intermediate results.

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<`$L_0, L_1, ...$`>`, where L_i is the shape of cell i. Note that if the list is the representation of an SQL query, list cells are always vectors, as they typically represent collections of columns or row indices.

Built-in Function in Groups

Built-in functions are categorized based on their pre-defined 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`, which originate from aggregation functions in SQL queries;

Scan (S): boolean selection function `compress`. After the selection, it retrieves 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.

7.2.3.2 Shape Propagation Rules

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

Vector Functions

For vector functions, the return shape can 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.

- **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 7.2 presents the shape propagation rules for binary element-wise functions.

Statically known vector lengths provide exact shape propagation rules and can throw errors at compile time. If one or more arguments are of 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.

Table 7.2 – 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

- **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 7.3 presents the rules for unary element-wise functions.

Table 7.3 – Rules for unary element-wise functions (E)

$F_U(x)$	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
Return	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)

- **Reduction functions.** Reduction functions take a vector as input and compute a scalar output value. In all cases this produces a $V(1)$ shape described in Table 7.4.

Table 7.4 – Rules for reduction functions (R)

F_R(x)	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
Return	V(1)	V(1)	V(1)	V(1)

- **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 7.5 describes the full shape rules where `x` is the mask and `y` the values vector.

Table 7.5 – 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				

For constant sized vectors where the lengths agree, 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. Figure 7.7 shows an example of two vectors which have the same output scan shape.

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

Figure 7.7 – 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. Table 7.6 shows the rules for the array indexing function where `x` contains the values, and `y` the indexes to fetch.

Table 7.6 – Rules for array indexing (X)

F_X(x,y)	x			
y	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
V(1)	V(1)	V(1)	V(1)	V(1)
V(c_1)	V(c_1)	V(c_1)	V(c_1)	V(c_1)
V(d_1)	V(d_1)	V(d_1)	V(d_1)	V(d_1)
V$_s$(a_1)	V$_s$(a_1)	V$_s$(a_1)	V$_s$(a_1)	V$_s$(a_1)

- **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. Full shape rules are found in Table 7.7, where `x` is the values and `y` the extra parameter.

Table 7.7 – Rules for special boolean functions (B)

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(1)	V(c_0)	V(d_0)	V$_s$(a_0)
V(d_1)	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)
V$_s$(a_1)	V(1)	V(c_0)	V(d_0)	V$_s$(a_0)

List Functions

As introduced in Section 4.4.2, HorseIR supports one `each` function with a unary operation, `each(f,x)`, which applies a function `f` over all elements of list `x`, and three `each` functions with a binary operation, `each_item(f,x,y)`, `each_left(f,x,y)` and `each_right (f,x,y)` where `f` is a binary function that takes two input values. For the binary operations, the i^{th} elements of the outputs are computed as `f(x(i),y(i))`, `f(x(i),y)`, and `f(x,y(i))` respectively.

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).

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 as described in Section 7.3.2.

7.2.3.3 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 independent statements. Two statements are in the same fusible section if they are conforming.

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. A fusible section will be translated into one coherent code snippet by the code generator as discussed in Section 7.3.

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 7.8. In our approach, we check conformability between statements and the *definition statements* of their input parameters (predecessors in the data dependence graph).

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 conforming with

Table 7.8 – Conforming 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 ×				

their uses. Each recursive call tree therefore forms a single fusible section that ends when no more statements may be fused. In addition to conformability, we ensure that reductions may only terminate fusible 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 Algorithm 1 and subsequent sections.

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 conforming with the shape of the definition statements according to Table 7.8.

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

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

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 Algorithm 1 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 fusible sections, a data dependence will exist between sections. This dependency would require an intermediate result to be stored, which negates the purpose of our approach. We therefore remove any statement whose successors are in different fusible sections from the fusible section.

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.

An Example

Given the HorseIR program in Figure 7.2b, our algorithm identifies a fusible section shown in Figure 7.8. Initially, the columns `t1` and `t2` are assigned a dynamic vector shape with a unique ID, `V(d)`, as the exact size depends on the input table. Next, the element-wise function in statements S_9 propagates the vector shape `V(d)` according to the shape rules. Both compression functions in S_{10} and S_{11} generate a new shared scan shape `Vs(t3)` as they both use the same boolean mask. The following binary function uses this equality to correctly infer its output shape. Finally, the

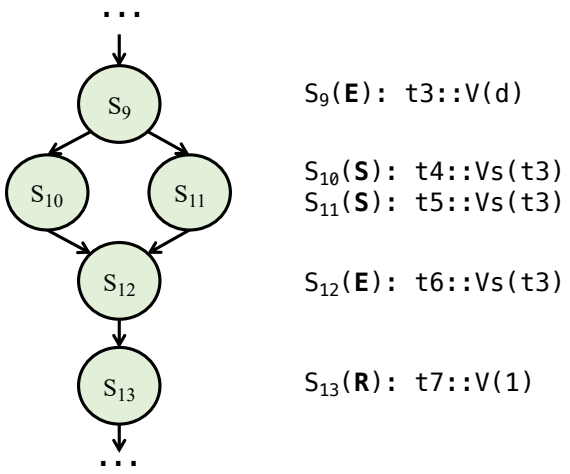


Figure 7.8 – A fusible section for the HorseIR program in Figure 7.2b. The text format on the right hand side is `<statement>(<group>): <variable>::<shape>`.

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 Figure 7.6 shows the generated C code for the complete fusible section. Note the if-condition for boolean selection and the accumulator for the reduction. The code generation strategy for fusible sections will be introduced in Section 7.3.1.

7.3 Code Generation Optimizations

In this section, we first identify fusible sections after the conformability analysis in Section 7.2.3.3, and generate fused code automatically in Section 7.3.1. HorseIR optimizer adopts a set of loop fusion based optimizations since loop fusion is a key optimization for array-based languages [24, 47], and it is similarly important for HorseIR. We then explore fusions/specializations based on patterns specific to HorseIR code in Section 7.3.2.

7.3.1 Automatic Loop Fusion

When generating code for a fusible section, first each fusible section is associated with a fusion node, the nodes are then optimized, and lastly the code is emitted.

7.3.1.1 Fusion Nodes

Each fusible 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.

7.3.1.2 Code Generation for Vectors

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

(a) Case 0

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

  z = z Rop expr_rhs;
}
z = Rfinal(z);
```

(b) Case 1

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

(c) Case 2

```
// reduction: NO
// boolean selection: NO

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

  z[i] = expr_rhs;
}
```

(d) Case 3

Figure 7.9 – 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. Figure 7.9 shows the variations of the code generation patterns.

Note that when generating parallel code for Figure 7.9c, 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.

7.3.1.3 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 7.10 – 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. Figure 7.10 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.

Name	Code Examples	Description
FP-1	`t0:? = @compress(mask, k0);` `t1:? = @compress(mask, k1);`	Materialize columns with the same condition

FP-2	`t0:? = @lt(k0, k1);`
	`t1:? = @compress(t0, k2);`
	`t2:? = @len(k2);`
	`t3:? = @vector(t2, 0:bool);`
	`t4:? = @index_a(t3, t1, 1:bool);`
		Semi-join with a selection condition

FP-3	`t0:? = @each_right(@index, k0, k1);`
	`t1:? = @each(@unique, t0);`
	`t2:? = @each(@len, t1);`
	`t3:? = @raze(t2);`
		Return the number of unique values after group by

FP-4	`t0:? = @index(k0, v0);` `t1:? = @index(k1, v0);`	Materialize columns with a common vector containing indices

Figure 7.11 – Code examples for fusing with patterns

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.

7.3.1.4 Further Fusion Opportunities

Fusible sections can be merged if: (1) they share the same loop head, and (2) there is no data dependency between the loop bodies. This is particularly useful 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 reduces the number of parallel synchronization barriers.

7.3.2 Fusing with Patterns

Fusing with fusion nodes is beneficial, but there are other optimization opportunities for common patterns of array operations that occur specifically in the code generated

for database operators. Therefore, a set of patterns (FP), identified and adopted for optimizing these situations, can be found in Figure 7.11. Patterns are designed for merging statements and guiding our optimizer to generate efficient C code.

FP-1 This is a common pattern for materializing multiple columns after a selection in SQL. As can be seen in Figure 7.12, this pattern tries to fuse two or more statements containing the boolean selection function `compress` with a common mask. The generated fused code has a single loop with one accumulator that reduces the cost of synchronization for multiple loops when parallel code is implemented.

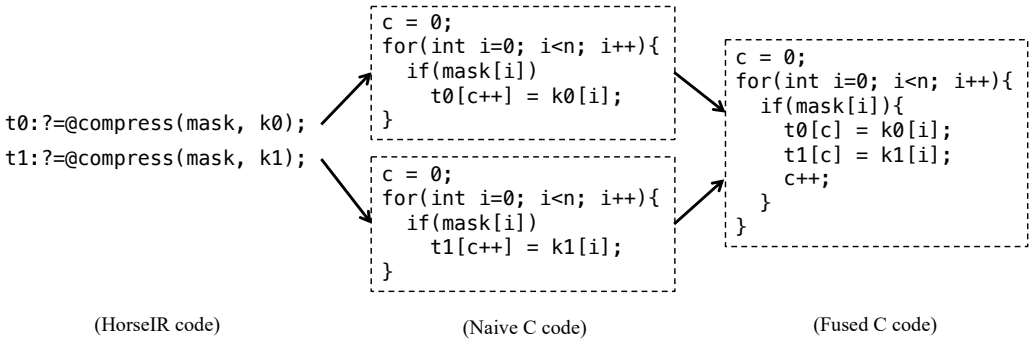


Figure 7.12 – Illustration for generating fused code with FP-1

FP-2 This is a HorseIR pattern that is the result of a SQL semi-join when a selection is applied on one column and it works with enumeration. Translating FP-2 in a coherent step to code results in fewer intermediate results, and thus, is more efficient than a general join implementation. In Figure 7.13, we can see that this pattern works when input vectors `k0`, `k1`, and `k2` have the same length. The comparison between `k0` and `k1` creates an intermediate vector `t0` which is later used in the boolean selection to create another vector `t1`. On the other hand, `t1` is used only once as the index in an array indexing function. Therefore, statements can be fused in a loop without actually generating `t0` and `t1`.

FP-3 SQL queries often group row based on a column and then perform aggregation per group. We discussed the generated SQL code in Section 5.1.1.5. The

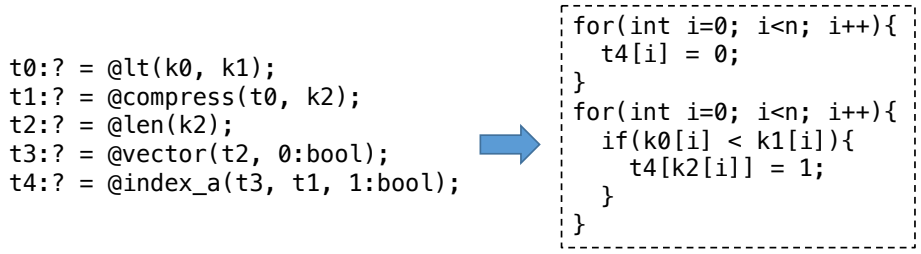


Figure 7.13 – Illustration for generating fused code for FP-2

aggregation often follows the pattern shown in Figure 7.11 and performs list indexing using `each` functions. In the example, there are two `each` functions with different aggregation functions: `@unique` returns the indices of the first unique items, and `@len` returns the length of an array. After the two aggregation functions, each cell in the list `t2` contains a single integer, indicating the number of unique items in each cell. Finally, the list is flattened by the function `raze` and a vector of integers is returned. All statements in this pattern can be efficiently implemented without allocating additional space to store intermediate results as depicted in Figure 7.14.

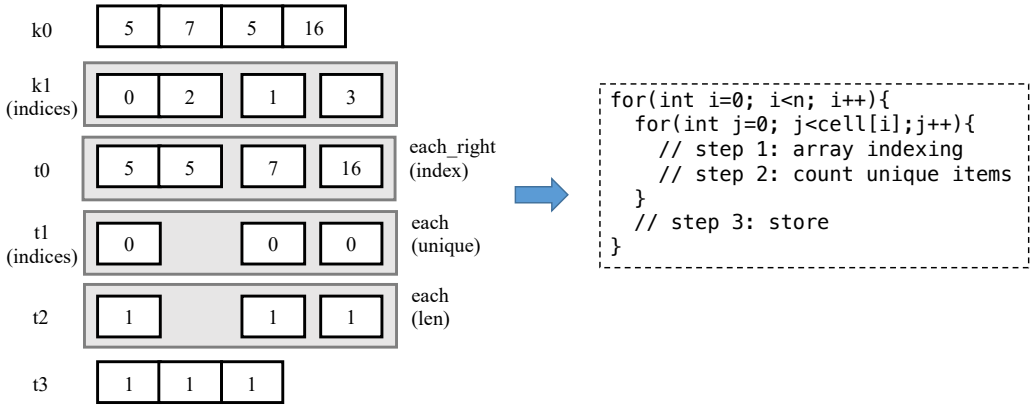


Figure 7.14 – Illustration for generating fused code with FP-3

FP-4 It is similar to the pattern `FP-1` to materialize columns with a boolean mask, but with indices. After a join or groupby operation, HorseIR returns indices (i.e., row ids) instead of actual values. Therefore, new columns are created by retrieving the actual values using the index vectors. Materializing multiple

columns with the same index vector can be done in a fused manner as illustrated in Figure 7.15.

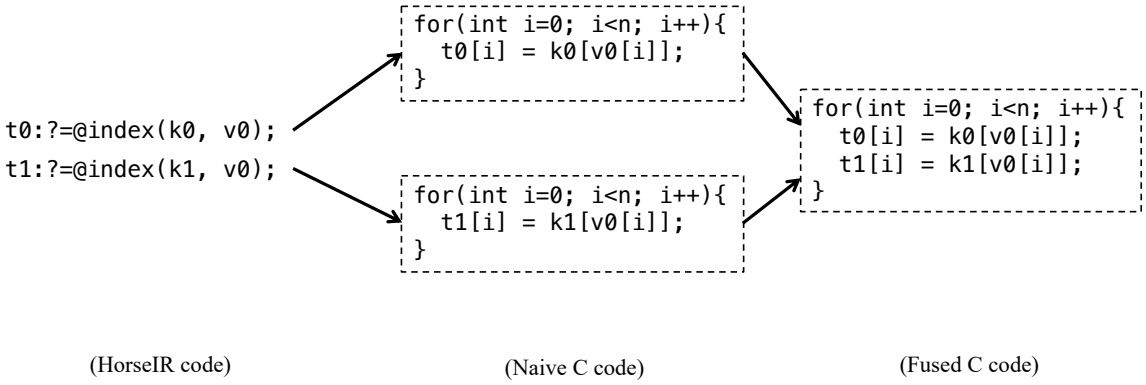
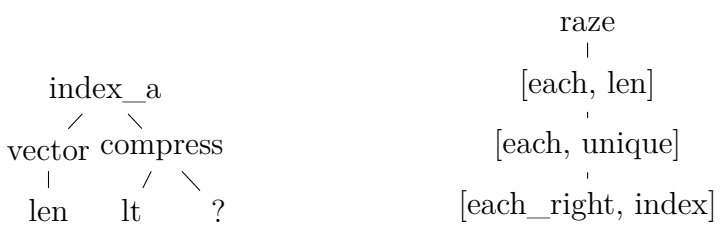


Figure 7.15 – Illustration for generating fused code with FP-4

Algorithm 2 shows how patterns can be matched in a HorseIR program to generate efficient code. There are two directions for matching: (i) *Top-to-Bottom* needs the information of how statements are used by looking up define-use chains (DUChain) for FP-1 and FP-4; and (ii) *Bottom-to-Top* requires the information of definitions of statements by searching in use-define chains (UDChain) for FP-2 and FP-3. Figure 7.16 shows that patterns are formed in a tree-based structure for Bottom-to-Top matchings. Once a pattern is matched, all matched statements need to be set visited and their information is saved for later code generation: (1) if a statement is only marked as visited, the statement is skipped; (2) if a statement marked as visited matches a Top-to-Bottom pattern and it matches the last statement of the pattern, all statements matched are fused; (3) if a statement marked as visited matches a Bottom-to-Top pattern and it matches the root of the pattern, it is translated with a prepared template to optimized C code; and (4) if a statement is not marked as visited, it is translated to an invocation to a library function which is written in C (this statement was not found to be fusible with anything).

Algorithm 2: Identify patterns for fusion

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



(a) FP-2 (b) FP-3

Figure 7.16 – Patterns designed for FP-2 and FP-3

7.4 Cross Optimizations

Considering SQL statements with embedded UDFs, as we have seen in Section 5.3, both the SQL and the UDF part are independently translated into HorseIR and the resulting code then merged to create a main method which calls the method representing the UDF. We discussed in Section 7.2 that we use method inlining to move all code into one method. In here, we provide more details of the motive behind this approach.

Our running example in Figure 5.22 shows the merged code with the clear separation of the SQL-based and the UDF-based parts. If we were to optimize both parts independently using loop fusion and pattern-based fusion as just discussed, the overall result would be sub-optimal. In fact, if we look at the dependence graph for this program on the left side of Figure 7.17 (with S_0 to S_4 depicting the statements in the code), we can see that the optimization opportunities are now separated into three snippets: before, after, and in the method being called in the statement S_3. The snippets have to be optimized individually because the content of the statement S_3 is invisible to the rest of the code. Thus, statements S_1 and S_2 of the main method need to be evaluated and intermediate results `t4` and `t5` cannot be eliminated as the method `calcRevenueChangeScalar` requires their actual values to be passed as parameters. Furthermore, the return value of the method needs to be materialized to be assigned to `t6` which is then the input of statement S_4. This means the potential scope for fusion is significantly reduced leading to more intermediate results.

Therefore, we do not optimize the individual parts independently, but aim at a holistic optimization. The idea to enable this cross-optimization using: *method inlining* as outlined in Section 7.1. As mentioned, this involves replacing the method calls within the main method by the corresponding code segments that constitute the method that is being called. This modified version of the HorseIR program will now have fewer method calls and is conducive to the fusion optimizations that we discussed in Section 7.3.

For our example program in Figure 5.22 this means the code of `calcRevenueChangeScalar` can be inlined into the main method with the generated HorseIR

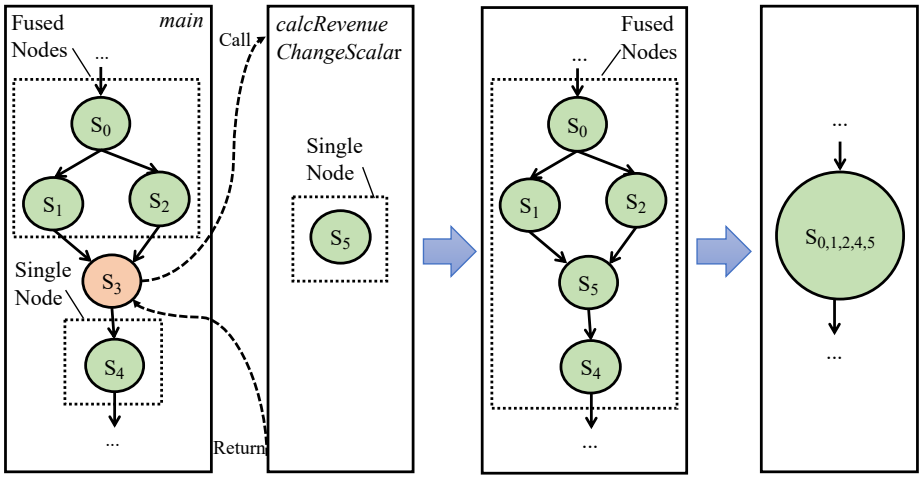


Figure 7.17 – Dependence graphs for the example in Figure 5.22 to show that method inlining helps explore more opportunities for automatic loop fusion.

being almost the same as the one in Figure 7.2 except for possibly different variable names. As a result, a dependence graph can be built across the main method, as illustrated on the right side of Figure 7.17, allowing for loop fusion across all statements and generating a single loop of all tasks. In particular, the boolean predicate statement (S_0), the compress statements (S_1 and S_2), the multiplication statement (S_5), and the reduction statement (S_4) can be fused together when generating the C code in Figure 7.18. The core computational logic of the program is now consolidated into a single loop which performs the tasks of selecting the relevant price and discount values as well as calculating the net revenue change. This code is efficient compared to the original version without method inlining, as it avoids the materialization of any intermediate results introduced by UDF invocations.

```
1 ...
2 revenue = 0;
3 for(i = 0; i < numRows; i++){
4   if(discount[i] >= 0.05)
5     revenue += extendedprice[i] * discount[i];
6 }
7 ...
```

Figure 7.18 – Generated C code after the cross-method optimization in Figure 7.17

Chapter 8
Evaluations

We design and conduct experiments for evaluating the performance of HorsePower in multiple ways as shown in Figure 8.1. We start this chapter by describing the experimental setup that we use for all our experiments in Section 8.1. From there, we analyze the performance of HorsePower while executing standard SQL queries for the TPC-H benchmark, comparing with MonetDB in Section 8.2. Next, we study the performance of HorsePower running standard array programs, comparing with MATLAB in Section 8.3. After that, we evaluate the performance of HorsePower when executing database queries with embedded UDFs in Section 8.4. Finally, we test the performance of HorsePower executing HorseIR code on GPUs in Section 8.5. Scripts and data used in our experiments can be found in our GitHub repository.[1]

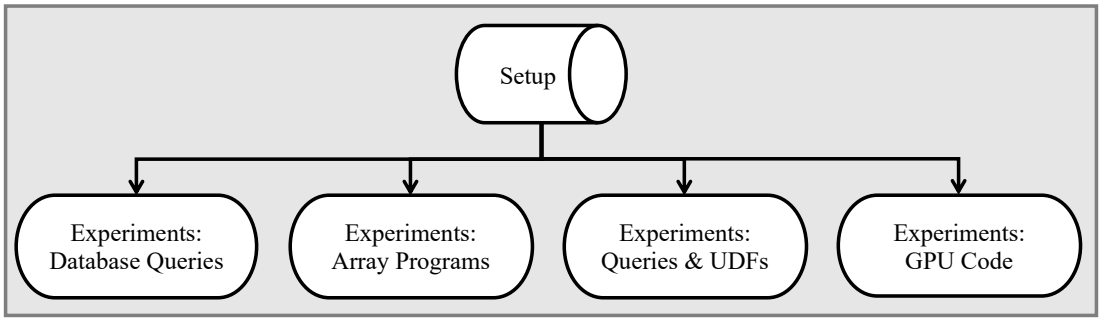


Figure 8.1 – Overview of experiments used to evaluate the performance of HorsePower

[1] https://github.com/Sable/chen-thesis-analysis

8.1 Experiment Setup

Item	**sable-intel**	**sable-tigger**
CPU	E7-4850 @ 2.00 GHz	i7-8700K @ 3.70 GHz
L1 Cache	32 KB	32 KB
L2 Cache	256 KB	256 KB
L3 Cache	24 MB	12 MB
Threads per Core	2	2
Cores per Socket	10	6
Sockets	4	1
Total Threads	80	12
Thread Scales	T1/2/4/8/16/32/64	T1/2/4/8/16
RAM Size	128 GB	32 GB
GPU	(No GPU)	GeForce GTX 1080 Ti
Operation System	Ubuntu 18.04.4 LTS	

Table 8.1 – Overview of machines used in experiments

Machines. Since the performance of HorsePower can be influenced by different hardware configurations, our experiments are conducted on two machines described in Table 8.1: *sable-intel* is a multi-socket multi-core server equipped with 4 Intel Xeon E7-4850 2.00GHz (total 40 cores with 80 threads) and 128 GB RAM running Ubuntu 18.04.4 LTS; and *sable-tigger* is a multi-core workstation equipped with Intel i7-8700K 3.70GHz (total 6 cores with 12 threads), an additional GPU (GeForce GTX 1080 Ti), and 32 GB RAM running Ubuntu 18.04.4 LTS.

Software. We use GCC v8.1.0 to compile C code with the 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. When compiling C-based GPU code on sable-tigger, we use NVIDIA PGI compiler 19.10-0 with the optimization option `-O4`.

Parallelism. In order to test the potential to exploit parallelism, we consider different thread scales for the two machines: T1/2/4/8/16/32/64 (T1 to T64) for

sable-intel and T1/2/4/8/16 (T1 to T16) for sable-tigger. These scales cover the number of physical cores each of the machines has.

Execution plans. As mentioned in Section 5.1.2, HorsePower supports translation from HyPer's optimized execution plans as well as MonetDB's non-optimized execution plans. We use the optimized execution plans from HyPer for all queries in the TPC-H benchmark in Section 8.2. On the other hand, as HyPer lacks facilities to generate execution plans with UDFs, we use the non-optimized execution plans of MonetDB to generate our HorseIR code in Section 8.4.

Methodology. In terms of execution time, we only consider the time once data resides in main memory. It is measured only for the core computation, and does not include the overhead for parsing SQL, plan generation, and serialization for sending the results to the client. We only consider core computation, as this is the main part where HorsePower provides new contributions. We consider performance variance by running each test 15 times, measuring the average execution time over the last 10 runs. After the first 5 runs, response time stabilizes, reducing the impact of caching and initial data transfer.

8.2 Experiments with a Database Query Benchmark

In this experiment we analyze how well HorsePower executes database queries originally written in SQL by comparing it to a state-of-the-art columnar RDBMS, MonetDB. Since MonetDB stores the columns of a table in contiguous space, its internal structure is similar to an array. Furthermore, we provide a detailed analysis of the impact of the compiler optimizations deployed.

8.2.1 Benchmark Overview

Our tests use TPC-H [85], a widely used SQL benchmark suite for analytical data processing. TPC-H mimics a Business to Consumer (B2C) database application. Its data is synthetically created and can vary in size. A *scale factor* (SF) of 1 corresponds

to a database size of approximately 1 GB, with higher scale factors proportionately increasing the size of the database. As can be seen in Figure 8.2, TPC-H has 8 tables with pre-defined primary/foreign key relationships in its schema. For instance, table `partsupp` has a foreign key `ps_partkey` referencing the primary key `p_partkey` in table `part`. The benchmark contains a suite of 22 SQL queries that range from simple queries to complex ones. Profiling data in Table 8.2 shows that these queries cover a variety of performance impacting dimensions such as the number of input tables, joins, condition predicates, aggregations, groupby operations, sorts, and output columns.

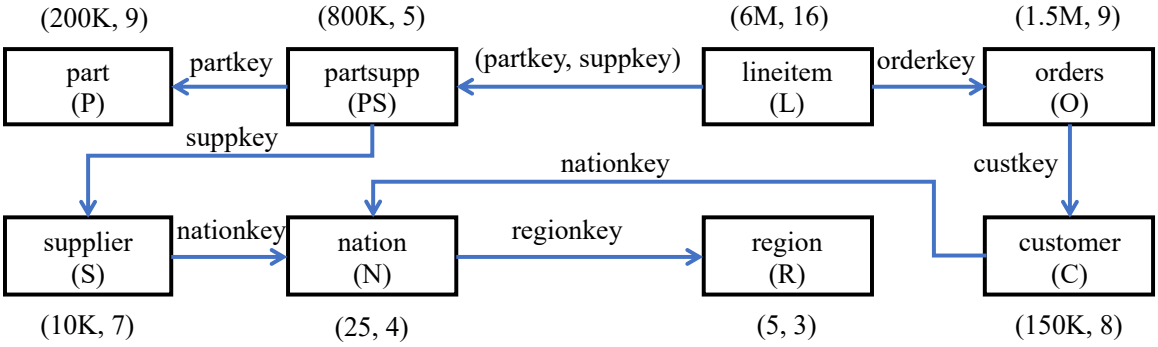


Figure 8.2 – TPC-H schema and its table sizes (row, column) on SF1.

8.2.2 Complete Suite Results

For each of the 22 queries of the TPC-H benchmark, we took the execution plan generated by HyPer and translated it to HorseIR as outlined in Section 5.1.2, followed by the compiler optimizations and code generation indicated in Chapter 7. We then ran the experiments using a scale factor of 1 (i.e., SF1) with both machines `sable-intel` and `sable-tigger` as shown in Table 8.1.

8.2.2.1 Results on sable-intel

Figure 8.3 shows the execution times of HorsePower and MonetDB with an increasing number of threads on `sable-intel`. We have the following observations.

ID	Tables	Join	Predicate	Aggregation	Groupby	Sort	Return
1	L	0	1	8	2	2	10
2	P,S,PS,N,R	8	13	1	0	4	9
3	C,O,L	2	5	1	3	2	4
4	O,L	1	5	1	1	1	3
5	C,O,L,S,N,R	5	9	1	1	1	2
6	L	0	4	1	0	0	1
7	S,L,O,C,N	5	9	1	3	3	8
8	P,S,L,O,C,N,R	7	10	1	1	1	5
9	P,S,L,PS,O,N	5	7	1	2	2	6
10	C,O,L,N	3	6	1	7	1	8
11	PS,S,N	5	6	2	1	1	3
12	O,L	1	5	2	1	1	3
13	C,O	1	2	2	2	2	4
14	L,P	1	3	1	0	0	1
15	S,L	2	2	2	2	1	5
16	PS,P,S	2	6	1	3	4	5
17	L,P	3	4	2	0	0	2
18	C,O,L	3	3	1	5	2	7
19	L,P	1	21	1	0	0	1
20	S,N,PS,P,L	4	9	1	0	1	5
21	S,L,O,N	5	13	1	1	2	4
22	C,O	2	6	3	1	1	7

Table 8.2 – Profiling data of all TPC-H queries: tables and the number of joins, predicates, aggregations, groupbys, sorts, and return columns

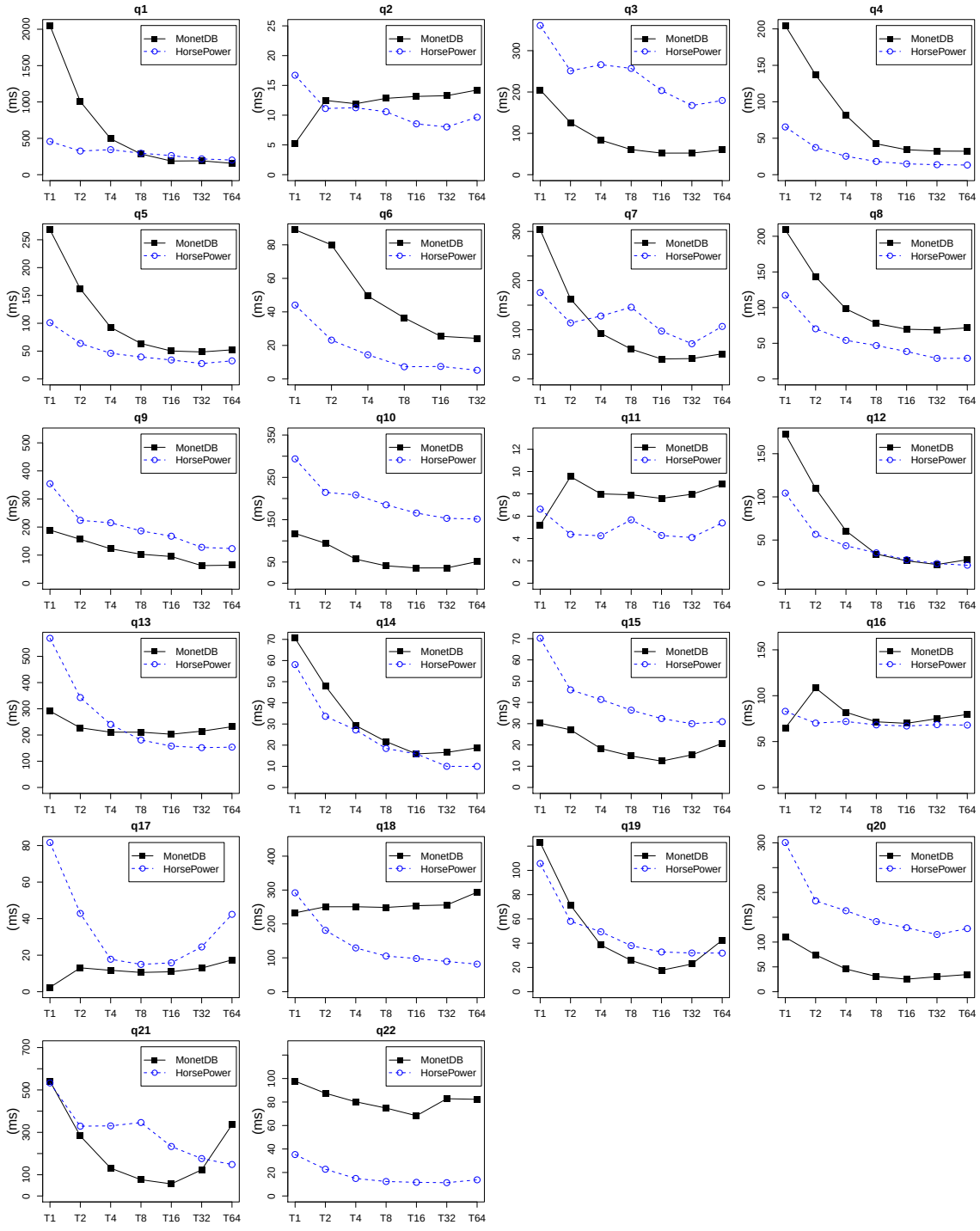


Figure 8.3 – (sable-intel) Performance comparison between MonetDB and HorsePower over all TPC-H queries with 1 GB input data (SF1)

Where we are clearly faster. HorsePower has distinct performance advantage over MonetDB in queries 4, 5, 6, 8, and 22. The lines for HorsePower are always clearly below the lines for MonetDB.

Queries 4, 6, and 22 benefit from the fusion of multiple statements involving element-wise functions and fusion patterns. In particular, the main computation of query 6 is fused into a single loop that greatly reduces the cost of intermediate results.

Queries 5 and 8 benefit from our efficient array-lookup join, which is faster than a general radix-based hash join when one column contains positive unique integers.

Where we are faster for most thread numbers. HorsePower is overall faster than MonetDB in queries 1, 2, 11, 12, 14, 16, and 18, but the difference depends on the number of threads.

In queries 2, 11, 16, and 18, MonetDB has the best performance using a single thread and performance gets worse with more threads. That is, it suffers from poor parallelism. In contrast, HorsePower can exploit parallelism and improve performance with increasing thread numbers. Thus, while it has worse performance than MonetDB with one thread, it outperforms MonetDB with multiple threads.

On the other hand, in queries 1, 12, and 14, HorsePower performs better than MonetDB at small thread numbers, but both have similar performance as the number of threads increases. That is both MonetDB and HorsePower can exploit parallelism. But on top of this, HorsePower can also perform well without parallelism due to its compiler optimizations.

Where performance varies. In queries 7, 13, and 19, performance varies depending on the number of threads.

For query 7, HorsePower is better than MonetDB for small number of threads, but worse for large number of threads. Query 7 has a join operation which dominates the performance of the query. The implementation of the join operation

follows the array-lookup strategy as mentioned in Section 6.3.2. However, this implementation cannot exploit parallelism well because the array construction for lookup is expensive due to the large array size. However, it achieves good performance already with a small number of threads. In contrast, MonetDB's join implementation can take advantage of parallelism but is unable to perform well with a few threads. As far as we can see, the performance differences are not due to optimization techniques, but are related to the particularities of the join implementation.

Query 13 has a performance bottleneck in the `like` function, which is used for string pattern matching. We adopted a simple algorithm tailored for SQL like functions. We materialize the result of the `like` function, which exposes great parallelism but still ends up slower than MonetDB with a small number of threads.

For query 19, MonetDB can first take advantage of more threads, outperforming HorsePower between 4 and 32 threads, because HorsePower contains more serial code. However, MonetDB's implementation has some performance issues with a large number of threads that HorsePower does not exhibit.

Where we are much slower. HorsePower is slower than MonetDB in queries 3, 9, 10, 15, 17, and 21. Execution time of these queries are dominated by heavy joins and groupby with little potential for compiler optimizations, and MonetDB's implementations for these database operators are generally more sophisticated than the limited implementations we so far have developed for HorsePower. We believe that by offering more specialized implementations, as most database systems do, we can close this performance gap.

In query 3, the input size for the second `groupby` operation is about 3.2M and its output size is about 830K. HorsePower so far does not fuse consecutive `groupby`. Thus, it is expensive to write out the output tuples after the second `groupby`.

In query 9, there is a four-column equal join on two columns with sizes 43K and 6M, and its result contains 320K rows. The performance of our strategy

for multiple-column joins depends on the selection of the first column, which is sub-optimal for this particular query.

In query 10, the most expensive part is the `groupby` operation on seven columns with types: <integer, symbol, floating, string, symbol, string, string>. We initially employed an *sort-and-scan* strategy that performs a linear scan after sorting. We improve this by 1.6 times over by using hash-based `groupby` with the general-purpose hash function MurmurHash3 [13]. It is still slower than MonetDB, but we think HorsePower can be extended with further implementations that can improve performance.

In query 15, the early materialization with the `compress` functions on three columns takes about 45% of the whole query time in HorsePower. Putting the materialization later would potentially improve performance. Thus, here is something for which our compiler could potentially be extended.

In query 17, MonetDB has an extremely fast performance for one thread and HorsePower approaches this value but with more threads. The query has two joins with two input columns with row sizes 204 and 6M, but identical. That means one join could be removed. However, our optimizer is unable to handle this case currently. In our future work we hope to provide a smarter optimizer for removing such kind of redundant code.

Query 21 is similar to query 19 but with a bigger fluctuation in the cases of 4, 8, and 16 threads. When looking into query 21, we identify expensive multiple-column joins with equal and non-equal operations. We first join the equal column and then the non-equal column, and then introduces intermediate results between the two operations, which is sub-optimal.

Preliminary Summary of Observations

Generally, for both HorsePower and MonetDB, the result of most queries follows the pattern that the performance first improves with increasing number of threads and later fluctuates with more threads. In HorsePower, the result of all queries has this trend. However, MonetDB has a couple of outlier queries where the performance

of a single thread is the best or significantly faster than two threads. Thus, MonetDB likely deploys a strategy that treats serial and parallel cases differently. Our implementation may introduce overhead with a single thread as it considers parallel constructs and algorithms, which aim at achieving better performance with more threads. For instance, in query 11, MonetDB has the best performance with one thread and no further improvement with more threads. However, HorsePower gains better performance as the number of threads increases. More importantly, the best configuration for HorsePower is faster than MonetDB in query 11. Furthermore, it is more common in MonetDB than HorsePower where performance gets worse for a large number of threads. This means it is different for MonetDB to configure the system, i.e., choose the number of threads as the optimal number depends on the query. In contrast, for HorsePower, choosing a relatively large number of threads for all queries is a safe choice.

In general, we observe that about half of the 22 queries benefit from HorsePower's compiler-based optimizations. However, while HorsePower already has a relatively efficient implementation for the most costly database operations, improvements are still needed where these operators make up the largest part of the execution.

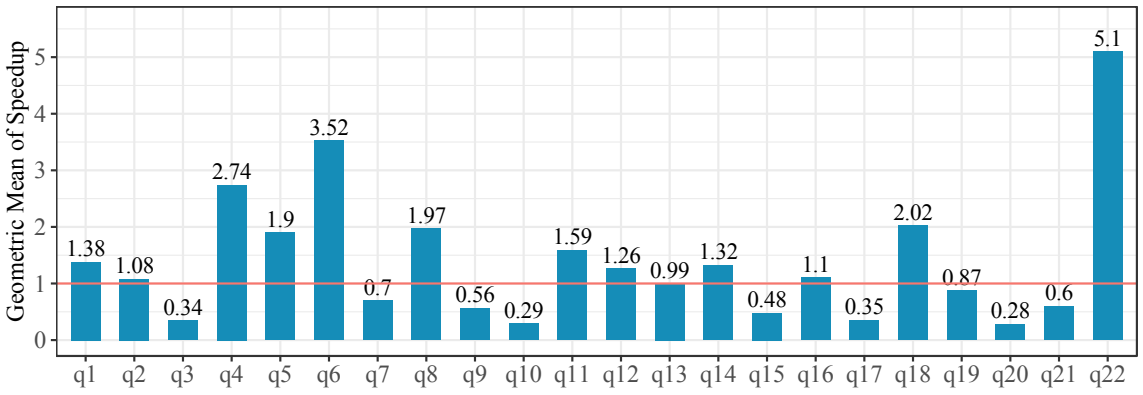
Geometric Mean

In order to have a clearer quantitative performance comparison between HorsePower and MonetDB, we use the geometric mean.

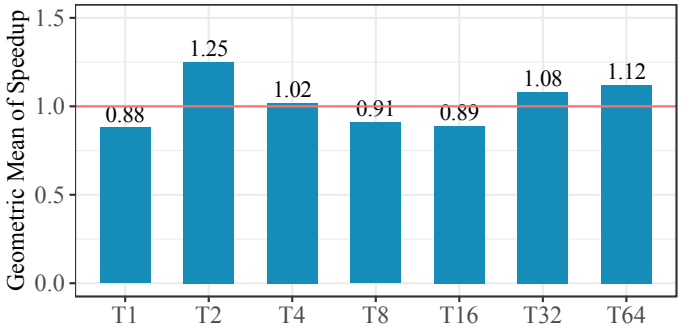
Figure 8.4a shows the results per TPC-H query. That is, for a given query, if the execution times for each of the n threads for MonetDB and HorsePower are $(m_{t_1}, m_{t_2}, \ldots, m_{t_n})$ and $(h_{t_1}, h_{t_2}, \ldots, h_{t_n})$, respectively, then, we calculate the geometric mean for this query as:

$$GeoMean(Query) = \sqrt[n]{\frac{m_{t_1}}{h_{t_1}} \cdot \frac{m_{t_2}}{h_{t_2}} \ldots \frac{m_{t_n}}{h_{t_n}}} \tag{8.1}$$

Similarly, Figure 8.4b shows the results per thread. That is, for a given number of threads, if the execution times for each of the n=22 queries for MonetDB and HorsePower are $(m_{q_1}, m_{q_2}, \ldots, m_{q_n})$ and $(h_{q_1}, h_{q_2}, \ldots, h_{q_n})$, respectively, then we calculate



(a) Geometric mean as a measure of speedup HorsePower vs. MonetDB for the 22 TPC-H queries



(b) Geometric mean as a measure of speedup HorsePower vs. MonetDB for each thread level

Figure 8.4 – (sable-intel) Results of individual queries and threads in terms of geometric mean.

the geometric mean for this thread level as:

$$GeoMean(Thread) = \sqrt[n]{\frac{m_{q_1}}{h_{q_1}} \cdot \frac{m_{q_2}}{h_{q_2}} \ldots \frac{m_{q_n}}{h_{q_n}}} \tag{8.2}$$

Values greater than 1 mean that MonetDB takes overall longer to execute queries than HorsePower. Thus, we can use this geometric mean as a measure of the speedup of HorsePower over MonetDB. As can be seen from Figure 8.4a, 10 queries are faster in HorsePower than MonetDB, 4 queries are close, and 8 queries are slower. From Figure 8.4b, we can see that HorsePower achieves better performance on T2/4/32/64, while MonetDB has the best performance on T1.

8.2.2.2 Results on sable-tigger

The overall performance comparison between HorsePower and MonetDB on the TPC-H benchmark on `sable-tigger` can be found in Figure 8.5 (detailed response time per query and thread) and Figure 8.6 (geometric means). Compared with the results on sable-intel in Section 8.2.2.1, sable-tigger has much better response times for both HorsePower and MonetDB because its single core is more powerful than sable-intel. In Figure 8.6a, we can see that HorsePower executes **15** queries faster than MonetDB, 1 query in almost the same time, and 6 queries slower. When we look into the detailed results of Figure 8.5, we can see that the queries which are faster on both sable-tigger and sable-intel have similar performance trends as described in Section 8.2.2.1. However, there are a couple of queries which used to be significantly slower than MonetDB on sable-intel, but are faster on sable-tigger (queries 7, 15, and 19) or as fast on sable-tigger (query 9). As can be seen, the speedup change for query 9 is huge from 0.56 to almost 1. In Figure 8.6b, we observe that MonetDB achieves the best performance with a single thread, whereas HorsePower has better performance over every other thread level.

Since sable-tigger supports additional instructions for vectorization (i.e., AVX and AVX2) compared with sable-intel, we conduct additional experiments to compare two versions of HorsePower with such instructions enabled (by default) and disabled (`-mno-avx` and `-mno-avx2`). However, we cannot see any significant performance

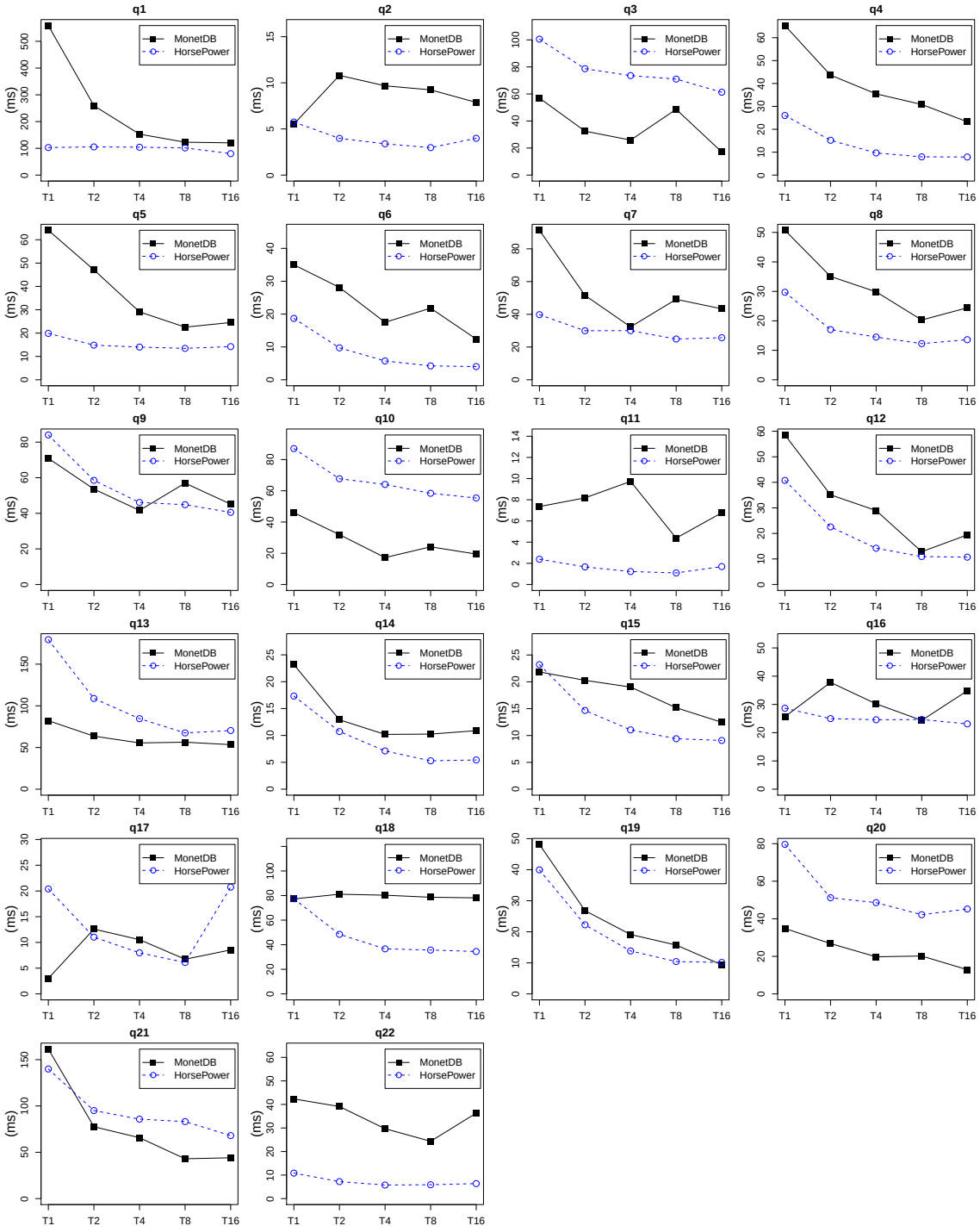
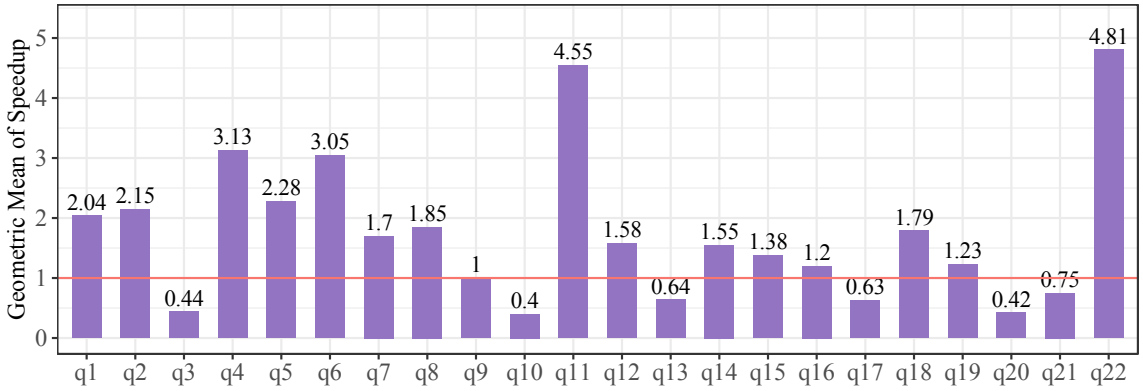
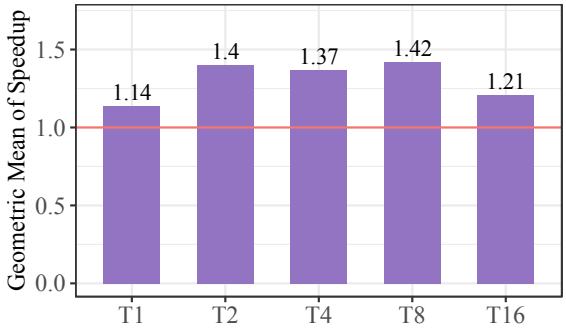


Figure 8.5 – (sable-tigger) Performance comparison between MonetDB and HorsePower over all TPC-H queries with 1 GB input data (SF1)



(a) Geometric mean of speedup over queries for TPC-H queries in Figure 8.5



(b) Geometric mean of speedup over threads for TPC-H queries in Figure 8.5

Figure 8.6 – (sable-tigger) Results of individual queries and threads in terms of geometric mean.

change between the two versions. That implies that our implementation underutilizes the hardware support for vectorization, given that options for SIMD instructions have been enabled in the C compiler to generate binary code. A potential future research could further improve the performance of the generated code by exploiting vectorization as outlined, e.g., by Peloton [57].

8.2.2.3 Discussion

Based on the results on the two machines, we can see that HorsePower is now on par with or better than MonetDB, a sophisticated database system that has been developed over decades. The optimizations in HorsePower's compiler are beneficial for many queries. Its implementations of database operators are already quite efficient but can still be improved. In addition, the performance of HorsePower can be affected by different hardware configurations. A query that is faster in HorsePower on sable-intel may be slower on sable-tigger, and vice versa.

Generally, MonetDB has better performance than HorsePower with a single thread. In fact, for some queries, MonetDB has the best performance with a single thread and more threads result in worse relative performance, such as in queries 3, 11, and 17. It is likely that MonetDB provides both serial and parallel versions of implementation and MonetDB's parallel version is slower for these queries, possibly because of data synchronization across threads. Another possible reason is that its implementation is quite efficient for serial code, but cannot be parallelized, and therefore, it requires a new parallelization strategy. Our HorsePower follows the design of array programming with fusion-based optimizations that exploits possible data parallelism in or between array-based operations. Performance generally improves with the number of threads and does not deteriorate with an increasing number of threads.

8.2.3 Effect of Optimizations

In order to study the effect of the different compiler optimizations implemented in HorsePower, we tested queries with four different optimization options: (1) No optimization (No-opt); (2) Automatic loop fusion only (FL-only); (3) Fusing with patterns

only (FP-only); and (4) All optimizations (All-opt). As can be seen in Section 8.2.2, the best performance for HorsePower is achieved with 16 threads on sable-intel and 8 threads on sable-tigger. Thus, to evaluate the impact of these optimizations under non-parallel and parallel environments, we tested with 1 and with 16 threads for sable-intel, and with 1 and with 8 threads for sable-tigger.

Table 8.3 – (sable-intel) Performance speedups on SF1 over No-opt obtained by various HorseIR optimizations for different queries.

Query	FL-only		FP-only		All-opt	
	1 th.	16 th.	1 th.	16 th.	1 th.	16 th.
q1	2.17	1.89	1.70	1.20	2.93	2.27
q2	1.10	1.00	4.10	131.5	7.85	182.0
q3	1.05	0.99	4.75	122.0	5.03	123.0
q4	1.08	1.19	2.15	6.38	2.58	6.70
q5	1.08	1.10	1.12	1.22	1.20	1.33
q6	8.44	8.45	1.04	1.19	8.00	7.17
q7	1.13	1.24	1.70	2.30	2.06	2.45
q8	1.09	1.06	1.26	1.46	1.38	1.41
q9	0.99	0.95	0.99	1.00	0.99	1.10
q10	1.05	0.99	1.60	8.02	1.61	8.35
q11	0.95	1.02	7.02	178.1	7.34	230.6
q12	2.13	1.22	0.98	1.04	2.15	1.22
q13	0.99	1.03	0.99	1.01	1.00	1.00
q14	1.32	1.14	1.37	1.65	2.07	1.59
q15	1.22	0.97	1.61	10.4	2.28	11.2
q16	1.05	0.99	1.53	13.5	1.66	13.8
q17	1.06	1.00	1.00	1.00	1.06	1.03
q18	0.97	1.07	9.02	517.3	9.02	530.6
q19	1.64	1.11	1.39	1.68	2.98	2.08
q20	0.96	1.00	3.76	121.0	4.17	129.3
q21	1.00	1.09	1.26	1.32	1.27	1.44
q22	1.02	1.14	1.01	1.07	1.01	1.14
GeoMean	1.26	1.20	1.79	5.62	2.41	6.74

Table 8.3 shows the speedup in execution time on sable-intel for the different optimized configurations compared to running with no optimization enabled. A first

observation is that the impact of these optimizations varies quite a bit among the different queries. For instance, for q2, FP has much more impact than FL, while the opposite is true for queries 1, 6, and 12. Even though q6 has only one fusion for element-wise and boolean selection functions, it is the longest fusion chain, which fuses 13 statements. Some queries, however, barely benefit from any optimization, such as queries 9, 17, and 22.

Using more than one thread to enable parallel execution is beneficial for most queries. However, in q16, there are many small cells (18314 cells, average size 6.5) and thus, our vector parallelization is underutilized. This is also the reason behind other queries with huge single-threaded speedups in which more threads result in much worse execution time. Computing the geometric mean over all the queries, the speedup is 2.41 for 1 thread between no and all optimization enabled, which is considerable, meaning the queries run in less than half the time with all optimizations enabled. The speedup is 6.74 for 16 threads meaning the more threads are enabled, the more can HorsePower benefit from the compiler optimizations.

Table 8.4 shows the speedup in execution time on sable-tigger with different optimizations enabled. Compared with sable-intel, the results show a similar overall trend in that `FP` has more impact than `FL`. The performance speedup for `FP` is less in sable-tigger because sable-tigger has a better non-optimized baseline. However, the speedup for `FL` remains almost the same on the two machines. In terms of geometric means for all queries over threads, the contribution of parallelism on sable-tigger has less impact than on sable-intel because sable-tigger is a newer machine with a faster CPU, and has less cores.

In summary, optimizations are critical to improving the performance of HorseIR. Fusion-based optimizations across statements are beneficial in many cases. However, there are two situations in which one has to be careful. Firstly, fusing all possible statements might become sub-optimal. Although this has not been observed in our work, a better strategy for loop fusion should consider multiple factors together, such as data locality, parallelism, and register pressure [78]. In practice, heuristics could 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

Table 8.4 – (sable-tigger) Performance speedups on SF1 over No-opt obtained by various HorseIR optimizations for different queries.

Query	FL-only		FP-only		All-opt	
	1 th.	16 th.	1 th.	16 th.	1 th.	16 th.
q1	1.64	1.74	1.64	2.20	2.62	2.85
q2	1.06	1.00	5.84	39.4	11.2	53.8
q3	0.97	1.05	8.17	34.3	8.47	38.6
q4	1.07	1.37	1.85	3.20	2.19	3.63
q5	1.03	1.05	1.05	1.07	1.41	1.21
q6	7.46	6.46	1.06	1.06	6.78	6.61
q7	1.20	0.81	2.33	1.41	2.86	1.77
q8	0.99	1.05	1.27	1.16	1.47	1.33
q9	1.05	0.95	1.04	0.97	0.90	1.07
q10	1.02	1.00	1.88	3.09	2.02	3.18
q11	1.01	1.01	10.5	66.2	11.4	74.3
q12	1.51	1.67	0.98	1.00	1.67	1.82
q13	1.00	1.05	1.00	1.04	1.03	1.09
q14	1.95	1.44	2.00	1.65	4.00	2.29
q15	1.36	1.10	1.89	3.76	3.03	5.02
q16	1.03	0.99	1.86	4.25	1.93	4.21
q17	1.09	1.07	1.01	0.99	1.08	1.07
q18	1.03	1.00	18.9	125.0	18.6	123.0
q19	1.61	1.56	1.39	1.52	2.87	2.91
q20	0.96	1.00	5.51	33.8	7.39	36.8
q21	0.99	1.26	1.46	1.56	1.52	1.64
q22	1.06	1.03	1.05	1.05	1.11	1.05
GeoMean	1.25	1.22	2.12	3.51	2.88	4.50

records qualify. In this case, the benefit of avoiding intermediate results is negligible, while the overhead of code fusion might become a factor. With additional data metrics, such unnecessary fusions could be avoided. Therefore, introducing runtime optimizations is an interesting avenue for future research. Future work will also look more closely at the fusion potential for join operations.

8.2.4 Scalability Study

In this section we look at the performance of HorsePower when data size increases. We have selected a subset of queries from the TPC-H benchmark, namely q1/4/6/12/14/16/19/22 for this experiments, omitting some of the queries where HorsePower still lacks efficient database operator implementations. We run these queries on sable-intel while increasing database scale factors from scale factor 1 to 16, while using T1/2/4/8/16/32/64 threads. Figure 8.7 shows that for both HorsePower and MonetDB, the geometric mean of the response times of all query/thread combinations for a given scale factor. Both HorsePower and MonetDB show excellent performance improvement in execution times when data size increases. In terms of performance, HorsePower is overall better than MonetDB. This shows that our approach can easily handle larger data sizes without any further tuning.

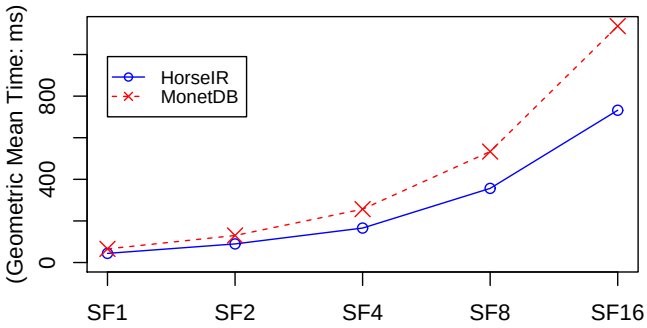


Figure 8.7 – Geometric mean execution time for HorseIR and MonetDB across five different SFs on sable-intel.

8.2.5 Compilation Time

There are a couple of stages from compiling query plans to generating output. We first compile query plans in a raw format to HorseIR code, then compile HorseIR code to C code, next compile C code to binary code, and finally execute the binary code. The first two compilation stages constitute a small portion of the whole pipeline since they are merely code transformations as described in Chapter 5 and Chapter 6. On the sable-intel machine, the compilation from the TPC-H query plans to HorseIR takes up to around 9 ms and the compilation from HorseIR to C takes an average of 4.6 ms. On the other hand, the compilation from C to binary code takes an average of 439 ms, that is about 97% of the whole compilation pipeline.

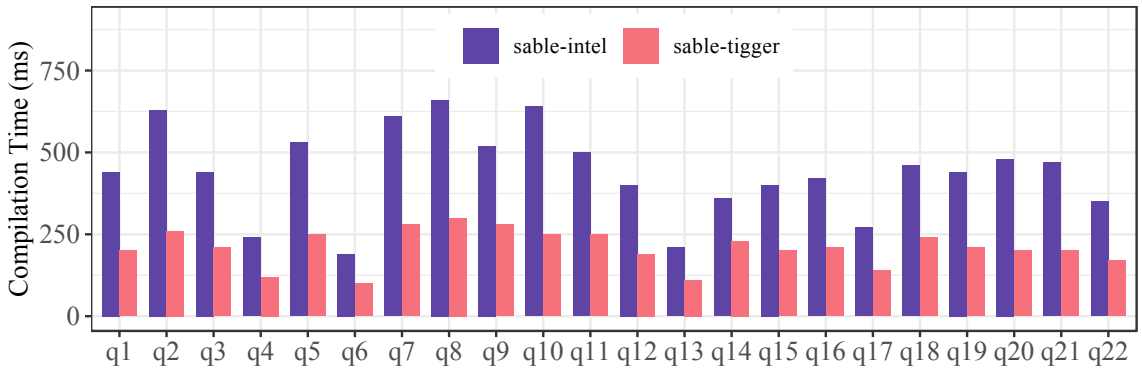


Figure 8.8 – Compilation time on both sable-intel and sable-tigger for TPC-H queries

Figure 8.8 presents this last step, i.e., the compilation time from the generated C code to binary code, including the linkage to the pre-built libraries, on both sable-intel and sable-tigger with the maximum compilation optimization `-O3`. Since the compilation process only uses a single thread, sable-tigger is faster than sable-intel because the processor is more powerful.

Compiling C code to binary consists of three parts: compilation setup, parsing, and code optimizations. The first phase usually is fast. We present the times in the other two phases in Table 8.5. As can be seen, almost all time is spent on the two phases of parsing and optimizations. The time spent on parsing takes around

[2]This phase includes both code optimization and generation. The optimization takes most time.

Table 8.5 – Compilation time breakdown in seconds and percentage

Query	sable-intel		sable-tigger	
	Parsing	Optimization[2]	Parsing	Optimization
q1	0.13 (30%)	0.30 (68%)	0.09 (45%)	0.11 (55%)
q2	0.14 (22%)	0.48 (76%)	0.09 (35%)	0.17 (65%)
q3	0.13 (30%)	0.30 (68%)	0.08 (38%)	0.13 (62%)
q4	0.13 (54%)	0.11 (46%)	0.08 (67%)	0.04 (33%)
q5	0.14 (26%)	0.39 (74%)	0.08 (32%)	0.17 (68%)
q6	0.12 (63%)	0.06 (32%)	0.08 (80%)	0.02 (20%)
q7	0.14 (23%)	0.46 (75%)	0.08 (29%)	0.20 (71%)
q8	0.15 (23%)	0.50 (76%)	0.09 (30%)	0.21 (70%)
q9	0.14 (27%)	0.37 (71%)	0.11 (39%)	0.16 (57%)
q10	0.15 (23%)	0.48 (75%)	0.09 (36%)	0.16 (64%)
q11	0.14 (28%)	0.35 (70%)	0.11 (44%)	0.13 (52%)
q12	0.13 (32%)	0.26 (65%)	0.08 (42%)	0.11 (58%)
q13	0.13 (62%)	0.07 (33%)	0.08 (73%)	0.03 (27%)
q14	0.13 (36%)	0.23 (64%)	0.14 (61%)	0.08 (35%)
q15	0.14 (35%)	0.26 (65%)	0.10 (50%)	0.09 (45%)
q16	0.13 (31%)	0.28 (67%)	0.08 (38%)	0.12 (57%)
q17	0.12 (44%)	0.14 (52%)	0.08 (57%)	0.06 (43%)
q18	0.14 (30%)	0.31 (67%)	0.09 (37%)	0.14 (58%)
q19	0.14 (32%)	0.30 (68%)	0.09 (43%)	0.11 (52%)
q20	0.13 (27%)	0.34 (71%)	0.08 (40%)	0.12 (60%)
q21	0.14 (30%)	0.32 (68%)	0.08 (40%)	0.12 (60%)
q22	0.13 (37%)	0.21 (60%)	0.09 (53%)	0.08 (47%)
Average	0.14 (34%)	0.30 (64%)	0.09 (46%)	0.12 (53%)

one third to half of the total compilation time. If we compare the numbers with the execution times in the Section 8.2.2.2 for a 1GB database, they are quite expensive. However, if queries are run many times, then this compilation time will be amortized. Moreover, looking at the execution times on larger datasets, as depicted in Section 8.2.5 we can see that compilation times play a much less role with large database sizes. Furthermore, the phase of optimizations could be further tuned by using customized optimizations with a reduced number of optimization passes while achieving potentially similar performance. Finally, instead of using C as our target code, we could consider lower-level code, such as LLVM [64], which would need less time to be compiled to binary than C code.

8.3 Experiments with an Array Language Benchmark

In order to understand the performance implications of using HorsePower for executing non-SQL based data analytics, we use the `Black-Scholes` algorithm from the PARSEC benchmark suite v3.0 [14], and the `Morgan` algorithm [18]. In this experiments, we analyze how well HorsePower performs in executing code originally written in MATLAB. We run all experiments on the server `sable-intel`. The description of these algorithms can be found below:

Black-Scholes: The Black-Scholes benchmark is used in finance to compute the price variation of European options over time by using a partial differential equation (PDE). This algorithm is fully vectorizable, and can be efficiently written using array programming. The data can be represented as one table with 9 columns. As the original implementation[3] is in C, we reimplemented this algorithm as MATLAB functions.

Morgan: The `Morgan` algorithm is also from a financial application. It contains a main function `morgan` and another function `msum`. The entire algorithm can be expressed in array operations without using any control structures. Instead

[3] The complete PARSEC package is downloadable here http://parsec.cs.princeton.edu

of using matrices as input as is done in its original implementation, we adapt it to using vectors which can represent two columns from one table. A third parameter which contains a single integer controls the size of the input columns for computation. We have set it to 1000. Since the original implementation is in APL, we reimplemented this algorithm as MATLAB functions.

Recall that HorsePower first translates MATLAB code into TameIR using the McLab framework, then translates the TameIR program into HorseIR, and then generates low-level C code that is finally compiled to binary execution. In our experiments, we first execute the original MATLAB program using the MATLAB interpreter with default settings. Secondly, we compile the HorseIR program, generated from the MATLAB code, to C code without any of the optimizations that we mentioned in Chapter 7, such as loop fusion. We refer to this C code version as *HorsePower-Naive*. As such, it is likely to produce a similar number of intermediate results as the MATLAB interpreter. Thirdly, we compile the HorseIR program into C code with all optimizations enabled. We refer to this C code version as *HorsePower-Opt*.

8.3.1 Experiment Results

Black-Scholes Results

Table 8.6 shows the execution times for MATLAB and the two HorsePower versions with different sizes of the Black-Scholes table (from 1 to 8 million rows). We also indicate the speedup of HorsePower over MATLAB in execution time for both HorsePower versions. Note that the MATLAB interpreter does not allow control of the number of threads and instead aims at using all physical threads (i.e., 40 threads on sable-intel). For a fair comparison, the shown HorsePower results are also with 40 threads.

We observe that the execution times for MATLAB and HorsePower-Naive are similar, with slightly better performance for MATLAB. Since there is no explicit loop in this benchmark, the effect of just-in-time compilation in MATLAB is negligible. We believe that MATLAB benefits from having more efficient library functions that work

Table 8.6 – Speedup of HorsePower over MATLAB in execution time using Black-Scholes (in milliseconds)

Size	MATLAB	HorsePower			
		Naive	Speedup	Opt	Speedup
1M	60.7	65.9	0.92x	6.5	9.34x
2M	145.0	136.5	1.06x	14.3	10.2x
4M	491.3	463.4	1.06x	48.5	10.1x
8M	1008.5	1384.1	0.73x	117.3	8.60x

well even in an interpreter mode. In contrast, MATLAB is significantly slower than HorsePower-Opt. The reason is that HorsePower-Opt optimizations, in particular loop fusion, are able to avoid many intermediate results, speeding up the computation by an order of magnitude. For both comparisons, the size of the data set plays a minor role.

Morgan Results

Table 8.7 shows the execution times for MATLAB and the two HorsePower versions with different sizes of the Morgan table (from 1 to 8 million rows). We also indicate the speedup of HorsePower over MATLAB in execution time for both HorsePower versions. For HorsePower the results shown are again with 40 threads.

Table 8.7 – Speedup of HorsePower over MATLAB in execution time using Morgan (in milliseconds)

Size	MATLAB	HorsePower			
		Naive	Speedup	Opt	Speedup
1M	80.7	104.7	0.83x	28.9	3.01x
2M	245.0	186.6	1.31x	58.0	4.23x
4M	479.8	378.1	1.27x	128.5	3.73x
8M	1488.3	905.3	1.64x	323.0	4.61x

For this benchmark, the naive version of HorsePower has already performance benefits compared to the MATLAB interpreter. HorsePower-Naive is slower than MATLAB code only with the smallest input size, but has already considerable speedup

with 2M rows which then further increases with large sizes. This implies that the code generated by HorsePower can achieve better parallelism for the built-in functions used in Morgan than MATLAB. Recall that the compiler optimizations that would fuse statements are not activated in HorsePower-Naive. Using HorsePower-Opt, the speedup ranges from 3 to 4.6. Having a close look at the code, we observe that loop-based fusion for fusing built-in functions plays a key role in the performance speedup. However, the overall speedup for HorsePower-Opt in Morgan is not as high as Black-Scholes.

Discussion

In summary, we can see that HorsePower is a promising approach to execute data analytics tasks in an efficient manner. This is due to its data-centric IR that makes it possible to exploit data-centric compiler optimization techniques. Even though the language syntax gap between MATLAB and HorseIR is non-negligible, HorseIR supports a relatively large set of core built-in functions which can be used in data analytics.

8.4 Experiments with a UDF Benchmark

In this experiments we compare the performance of HorsePower and MonetDB in executing SQL statements with embedded UDFs. We look at two benchmarks for that purpose. In Section 8.4.1, we look at a modified TPC-H benchmark, proposed by Froid [73], that rewrites the TPC-H queries to contain UDFs. In Section 8.4.2, we present a set of SQL queries that we created and that integrate the Black-Scholes algorithm in form of UDFs. In HorsePower, the UDF is written in MATLAB, for MonetDB in Python using the NumPy library (as MonetDB does not support MATLAB UDFs), with an effort to have similar code within the UDF so that the execution for the individual UDFs are similar.

8.4.1 TPC-H with UDFs

Froid [73] proposed a whole range of queries derived from the TPC-H benchmark in which part of the SELECT or WHERE clauses are outsourced into a UDF. In all cases, these are scalar UDFs. Some of these UDFs have embedded SQL statements. However, the McLab framework that we use to translate MATLAB programs currently only supports pure MATLAB programs. Thus, we excluded those unsupported queries and present results only for queries q1, q6, q12, q14, and q19. We set the scale factor of the TPC-H benchmark to 8 (i.e., SF8) meaning the size of the database is around 8GB. Thus, we have relatively large inputs for the computation within the UDFs.

Table 8.8 – Execution times and speedup (SP) of HorsePower over MonetDB using the modified TPC-H benchmark with UDFs.

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

Table 8.8 shows the execution times of these queries with a different number of threads using HorsePower and MonetDB, and presents the speedup of HorsePower over MonetDB in execution time. For HorsePower the results show the execution times after all optimizations have been performed.

When first looking only at MonetDB we can see that execution times are relatively low for some queries and improve with an increasing number of threads considerably (q1 and q14), but are high for others with little benefit of parallelization (q6, q12, q19). The reason is that in these queries, the UDF is in the WHERE clause and MonetDB has to perform costly data conversion when sending the entire database columns as arrays to the Python interpreter in order to execute the UDF. MonetDB is able to use zero-copy transfer for data types where the database system uses the

same main-memory representation as Python. But for strings, it needs to convert the data to a different format as the database internal and the Python formats are incompatible. This data conversion seems to not be parallelized to multiple threads, making it the predominant factor of the execution. In q1 and q14, the UDFs are in the SELECT clause (where data sizes are smaller as they got reduced due to the selection that was already executed), and do not require any string conversions.

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

8.4.2 UDF Derived from Black-Scholes

With the purpose of studying the performance of queries with embedded MATLAB UDFs, we set up the benchmarks as follows:

HorsePower version. We translated the MATLAB implementation of Black-Scholes to HorseIR, and then integrated it with the HorseIR code for the SQL component of the query, thus having both the SQL and the analytics function in the same IR.

MonetDB version. We implemented Black-Scholes as a Python UDF, and wrote SQL queries to invoke these UDFs.

Table 8.9 – Black-Scholes execution time Python vs. HorseIR

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

The Python UDF is implemented with the NumPy library using the same array programming style as the MATLAB UDF. In our Black-Scholes benchmark, each array operation in MATLAB has an equivalent array operation in NumPy. Table 8.9 shows the execution time of the Black-Scholes benchmark for the dataset in this section both using a Python program and using HorseIR (both naive and optimized). Execution is in one thread because NumPy does not support multi-threading for operations in the benchmark. Similar to what we have seen with our analysis with MATLAB, a naive usage of HorseIR provides similar execution time to Python; performing optimizations achieves a speedup of approximately 2.

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

```
CREATE SCALAR UDF bScholesUDF(spotPrice, ..., optionType)
{
  import blackScholesAlgorithm as bsa
  return bsa.calcOptionPrice(spotPrice, ..., optionType)
};
```

Next, we implemented the solution as a *Table UDF*, which returns in table form the computed `optionPrice` along with the associated `spotPrice` and `optionType` which are columns from the original input table.

```
CREATE TABLE UDF bScholesTblUDF(spotPrice, ..., optionType)
{
  import blackScholesAlgorithm as bsa
  optionPrice = bsa.calcOptionPrice(spotPrice, ..., optionType)
  return [spotPrice, optionType, optionPrice]
};
```

In order to have a broad set of tests and comparisons, we first integrated these two UDF versions into a straightforward base query. From there we created three significant variations of this base query that had different columns in the `SELECT` and `WHERE` clauses. Further, for each of the variations, we modified the values associated

with the conditional predicates in the selection (`WHERE` clause), so that the selectivity varies between high, low, and medium. In a highly selective condition, only a few of the input records fulfill the condition and thus are in the output result. A query with low selectivity returns most of the input records. Thus, our entire test case consists of 10 queries.

Table 8.10 – Execution time and speedup (SP) of HorsePower (HP) compared to MonetDB (MDB) for variations in Black-Scholes.

UDF	Selectivity	Table UDF (ms)						Scalar UDF (ms)					
		T1			T64			T1			T64		
		MDB	HP	SP	MDB	HP	SP	MDB	HP	SP	MDB	HP	SP
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_med	50.9%	914.7	262.4	3.49x	794.2	12.2	65.0x	308.6	86.6	3.57x	272.2	2.51	108x
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_med	50.9%	912.5	8.24	111x	811.6	4.22	192x	13.4	8.20	1.63x	4.16	4.29	0.97x
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_med	49.5%	906.5	263.7	3.44x	831.6	13.3	62.7x	678.9	184.1	3.69x	631.6	11.3	56.1x
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

Table 8.10 shows the selectivity for variations derived from Black-Scholes, covering low, medium, and high selectivity, and the execution time and speedup of HorsePower compared to MonetDB for both table and scalar UDFs. The result for 1 thread (T1) and 64 threads (T64) are presented.

Base query. Figure 8.9 depicts the base query `bs0_base`, which selects all the data from the database table and passes it to the UDF and returns all the data produced by the UDF.

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

```
1  -- Base query, bs0_base, Scalar UDF
2  SELECT spotPrice, optionType,
3    bScholesUDF(spotPrice,...,optionType)
4    AS optionPrice
5  FROM blackScholesData;
6
7  -- Base query, bs0_base, Table UDF
8  SELECT spotPrice, optionType, optionPrice
9  FROM bScholesTblUDF
10   ((SELECT * FROM blackScholesData));
```

Figure 8.9 – Base queries with scalar and table UDFs

code.

However, the speedup with one thread is already significant. In fact, with one thread, the execution time with 249.8 ms is basically equivalent to executing the Black-Scholes algorithm alone, without the SQL part, which is 247.9ms as shown in Table 8.9. For MonetDB, executing the algorithm within an SQL statement in the form of a Python UDF is with 927.5 ms nearly double as long as executing the Python function in standalone mode with 577.4 ms. As the SQL part of this query is straightforward, the reason for this performance penalty in MonetDB must be the communication between its SQL engine and the Python UDF interpreter.

Variation 1. The first variation `bs1_*` applies a predicate condition on `spotPrice`, a column which is actually part of the input database table. Figure 8.10 shows the example queries of variation 1 with scalar and table UDFs, as well the high selectivity. 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, by discarding records from the input that do not fulfill the predicate condition and only execute the UDF on the records that qualify. In contrast, a system following an inefficient approach will first compute the UDF over all the input records before applying the predicate.

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

```
1  -- Query, bs1_high, Scalar UDF
2  SELECT spotPrice, optionType,
3    bScholesUDF(spotPrice,...,optionType)
4    AS optionPrice
5  FROM blackScholesData
6  WHERE spotPrice < 50 OR spotPrice > 100;
7
8  -- Query, bs1_high, Table UDF
9  SELECT spotPrice, optionType, optionPrice
10 FROM bScholesTblUDF
11   ((SELECT * FROM blackScholesData))
12 WHERE spotPrice < 50 OR spotPrice > 100;
```

Figure 8.10 – Example queries of variation 1 with scalar and table UDFs

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

Variation 2. In the next variation, `bs2_*`, the SQL does not include the computed column `optionPrice` in the final result. Figure 8.11 shows the example queries of variation 2 with scalar and table UDFs, with the high selectivity. A smart system should be able to analyze the semantics of the request and avoid processing the UDF all together. As can be seen in the performance numbers, HorsePower achieves only a speedup of at most 2x with one thread and at most 5.5x with 64 threads for the scalar UDFs, but has a scale-up of at least 83x with table UDFs, going up to over 1000x for one UDF with 64 threads.

We can see that MonetDB is able to do the optimization when the SQL query is using the scalar UDF, avoiding the computation of the `optionPrice` column that is

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

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

Figure 8.11 – Example queries of variation 2 with scalar and table UDFs

not included in the final result. Similarly, HorsePower, being an integrated system, can avoid the computation of `optionPrice` by using a backward slice. However, with a table UDF, MonetDB is unable to avoid this computation as there is no way for it to pass this optimization information to the UDF interpreter. On the other hand, HorsePower uses method inlining and backward slicing to remove this computation, offering a huge advantage.

Variation 3. The last variation, `bs3_*` applies a predicate condition on `optionPrice`. Figure 8.12 shows the example queries of variation 3 with scalar and table UDFs, with the high selectivity. 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.

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

```
1  -- Query, bs3_high, Scalar UDF
2  SELECT spotPrice, optionType
3  FROM (
4    SELECT spotPrice, optionType, bScholesUDF(spotPrice, ..., optionType)
       as optionPrice
5    FROM blackScholesData
6  ) AS tableBS
7  WHERE optionPrice > 15;
8
9  -- Query, bs3_high, Table UDF
10 SELECT spotPrice, optionType
11 FROM bScholesTblUDF
12   ((SELECT * FROM blackScholesData))
13 WHERE optionPrice > 15;
```

Figure 8.12 – Example queries of variation 3 with scalar and table UDFs

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

8.5 Experiments with a GPU Benchmark

In this section, we describe a set of experiments that evaluate the performance of the hybrid CPU and GPU code generated by HorseGPU. Since a GPU is a powerful accelerator for data analytics, we mainly focus on database queries with UDFs, especially with UDFs that represent analytics tasks (Section 8.4.2) as they contain more complex computations than the ones derived from pure database queries (Section 8.4.1). The UDFs of these queries contribute a large part to the execution time of the queries. Thus, it is worth exploring performance acceleration with the help of executing the UDFs on GPUs. In particular, we look at the MATLAB programs

Black-Scholes and *Morgan*, described in Section 8.4.2.

We ran all experiments on the machine *sable-tigger*. The size of input for the two benchmarks is 8 million, and we use an increasing number of threads (from 1 to 16). We observe an expensive one-time cost in initializing GPU devices that is excluded from our performance results. This one-time cost takes an average of 283 ms and 286 ms for Black-Scholes and Morgan, respectively.

Table 8.11 – Overview of the portions for VersionCPU and VersionGPU

Version	Name	Description
VersionGPU	*GPU:Host*	Computation on the CPU side
	GPU:Kernel	Computation on the GPU side
	GPU:Data	Data transfer time between the CPU and the GPU
VersionCPU	*CPU:Selected*	Selected computation for the GPU but on the CPU
	CPU:Base	Computation remained on the CPU side

It should be noted that we have the following distinct versions: (1) *VersionCPU* is a CPU-based version with C code annotated by OpenMP; and (2) *VersionGPU* is a hybrid version with C code annotated by OpenMP (for CPU) and OpenACC (for GPU). In order to compare the two versions, we further break down the execution time into smaller portions, as shown in Table 8.11. VersionGPU has three portions, namely the execution on the CPU (`GPU:Host`), the data transmission (`GPU:data`), and the execution on the GPU (`GPU:Kernel`). VersionCPU has two portions, namely the computation of the parts that are executed on the GPU for VersionGPU referred to as `CPU:Selected`, and the part that always stays on the CPU for both versions, referred to as `CPU:Base`.

We measure the execution time for all portions in VersionCPU and Version GPU. It should be noted that `CPU:Base` and `GPU:Host` are the same if the same number of threads is given and the same optimization strategy is applied. In addition, we consider both unoptimized (i.e., naive) and optimized code to see how our compiler optimizations can affect the decision to use GPUs.

Table 8.12 – Black-Scholes: Performance comparison between VersionCPU and VersionGPU for both the naive and optimized versions over multiple threads (1 to 16).

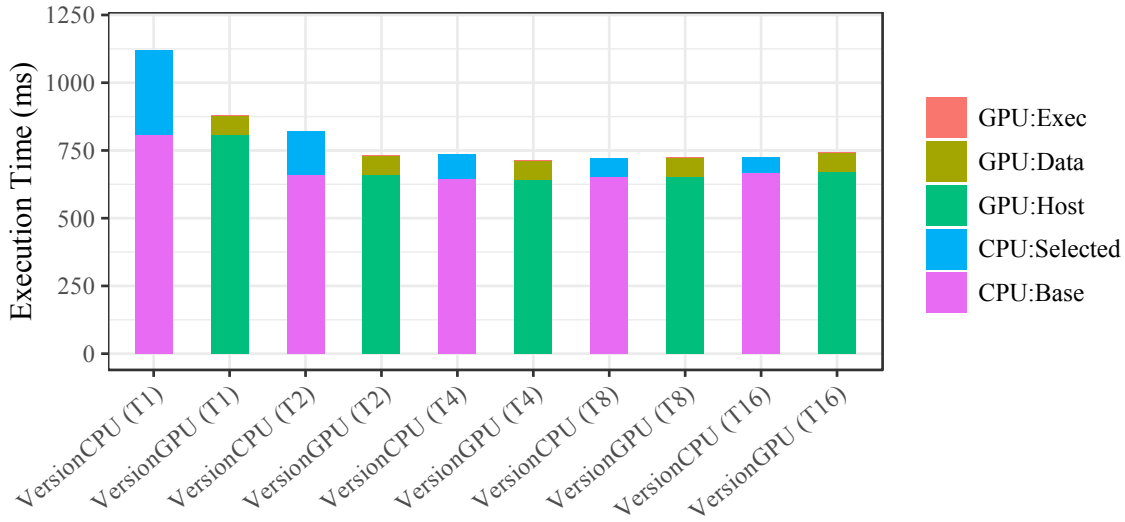
Size	Naive			Optimized		
8M	VersionCPU	VersionGPU	Speedup	VersionCPU	VersionGPU	Speedup
T1	1118	879.7	1.27x	332.6	175.1	1.90x
T2	819.1	733.1	1.12x	189.5	165.6	1.14x
T4	736.8	714.7	1.03x	89.9	174.2	0.52x
T8	722.7	726.4	0.99x	82.9	172.9	0.48x
T16	726.6	742.4	0.98x	69.9	183.6	0.38x

8.5.1 Black-Scholes Results

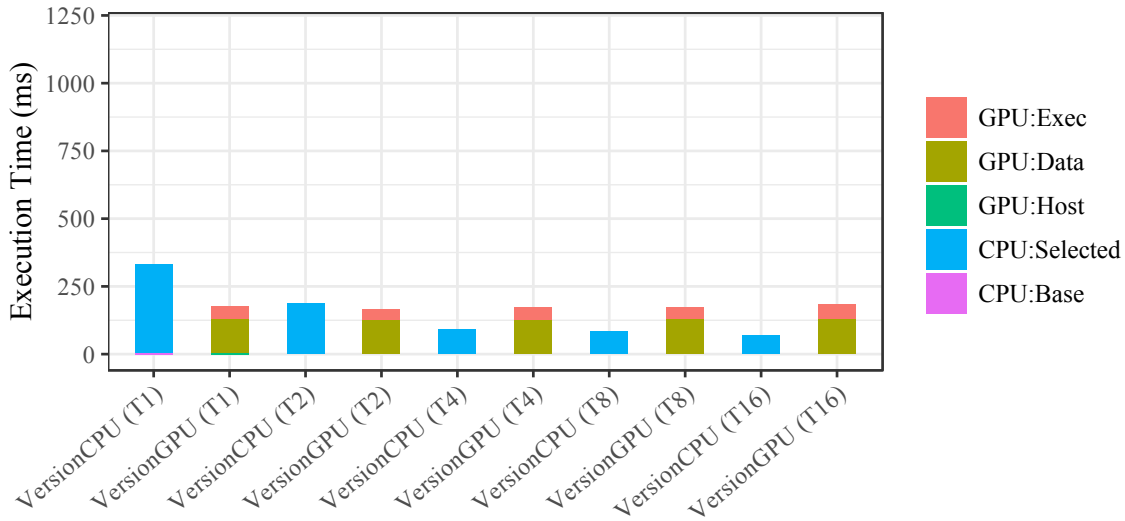
Table 8.12 presents the full execution time of Black-Scholes for both VersionCPU and VersionGPU on different configurations. Overall, VersionGPU has an advantage in the naive and optimized versions, but it becomes smaller with an increasing number of threads, and VersionGPU is worse than VersionCPU with many threads and optimizations enabled. By looking into the breakdown of the execution in Figure 8.13, we find that the naive and optimized versions have distinct results.

We can see in Figure 8.13a, for the naive version, around 28% of the computation of the VersionCPU is selected to be executed on GPUs once available, when there is one thread (T1). When using VersionGPU, the execution of this part is much faster, but data transmission takes time. In VersionGPU (T1), the GPU kernel execution time (GPU:Kernel) is about 3.6 ms while executing the same code on the CPU (CPU:Selected) is about 310.3 ms. As the number of threads increases, VersionCPU runs faster with the benefit of data parallelism, even though its performance on the selected portion is still slower than executing it on the GPU. This is due to the overhead in the data transmission that hinders the overall performance of VersionGPU. With 8 or more threads, the execution on the CPU is faster than outsourcing the compute-intensive parts to the GPU.

In Figure 8.13b, the selected portion (CPU:Selected), which is the one that is considered to be selected to be executed on the GPU, dominates the computation for VersionCPU (T1). We find that the computation in Black-Scholes can benefit from



(a) Black-Scholes: the naive version.



(b) Black-Scholes: the optimized version.

Figure 8.13 – Performance breakdown in Black-Scholes for both the naive and optimized versions.

our fusion-based optimizations that fuse many statements and generate fewer for-loops, thus avoiding intermediate results. This fused code is the one selected for the GPU. However, it leads to more data transmission between the CPU and the GPU. Therefore, once multi-threading is enabled, the CPU can be better: VersionCPU is faster and outperforms VersionGPU once there are 4 or more threads.

8.5.2 Morgan Results

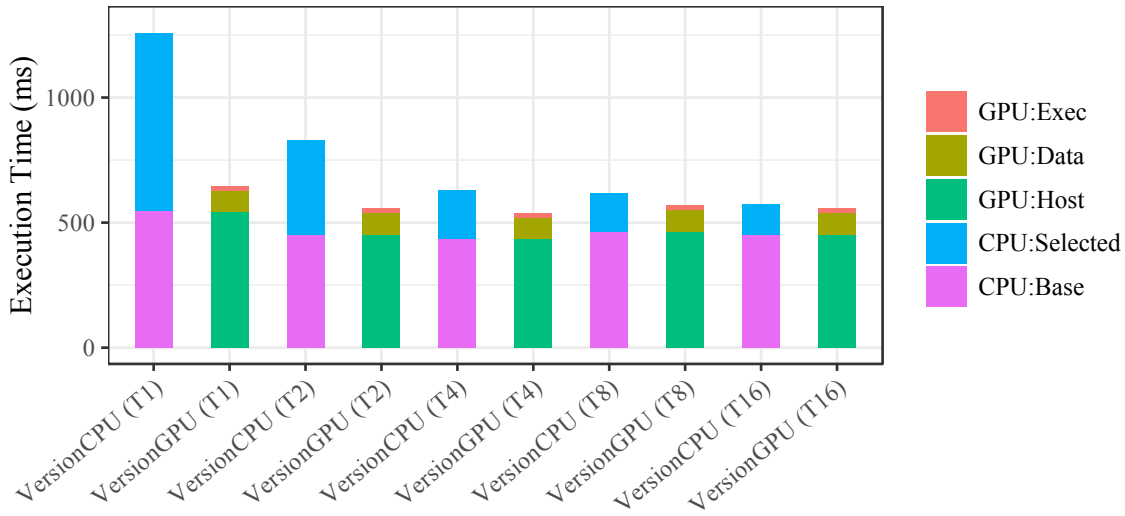
Table 8.13 – Morgan: Performance comparison between VersionCPU and VersionGPU for both the naive and optimized versions over multiple threads (1 to 16).

Size	Naive			Optimized		
8M	VersionCPU	VersionGPU	Speedup	VersionCPU	VersionGPU	Speedup
T1	1255	644.3	1.95x	353.9	368.3	0.96x
T2	828.6	555.7	1.49x	246.0	310.7	0.79x
T4	630.9	537.4	1.17x	209.1	291.9	0.72x
T8	617.8	567.3	1.09x	211.6	294.7	0.72x
T16	575.1	556.9	1.03x	210.0	296.8	0.71x

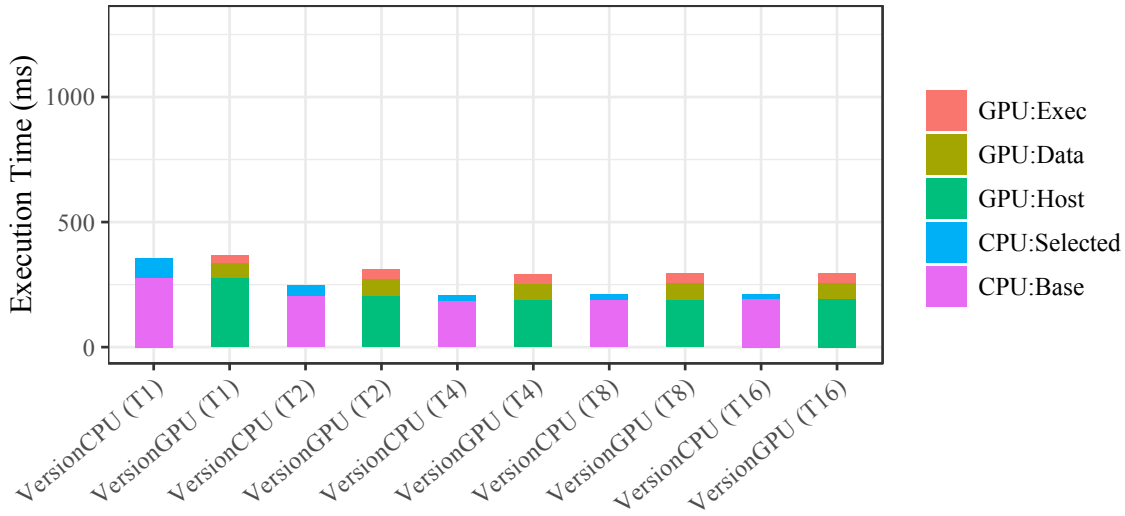
Table 8.13 presents the complete execution time of Morgan for both VersionCPU and VersionGPU under different configurations. Overall, VersionGPU performs better than VersionCPU for the naive version but it declines as the number of threads increases, and VersionGPU is always slower than VersionCPU in the optimized version. By looking into the breakdown of the execution in Figure 8.14, we observe that the naive and optimized versions have again quite different results.

Figure 8.14a shows the breakdown of the execution time for the naive version of Morgan. As can be seen in VersionCPU (T1), around 56% of the computation is selected (`CPU:Base`) and sent to the GPU for execution. The cost of data transmission is relatively small and the performance of GPU execution is significantly faster than CPU. By increasing the number of threads, the performance gap between `CPU:Selected` and `GPU:Data + GPU:Kernel` shrinks.

Figure 8.14b presents the breakdown of the execution time for the optimized



(a) Morgan: the naive version.



(b) Morgan: the optimized version.

Figure 8.14 – Performance breakdown in Morgan for both the naive and optimized versions.

version of Morgan. VersionCPU is faster than VersionGPU because of compiler optimizations, especially fusion-based optimizations. There are about six large vectors with almost 8 million floating numbers copied from or to the GPU. This overhead in data transmission takes the overall performance down, while VersionCPU is free from any data movement. Also, the execution of this highly fused code is only slightly faster on the GPU than the CPU with few threads and even worse with many threads. Thus, parallelism on the CPU works so well that it outperforms GPU execution.

8.5.3 Discussion

Based on the results of the two benchmarks, `Black-Scholes` and `Morgan`, we can see the advantage of having GPUs as accelerators as a means to improve the performance of data analytics functions. With the support of GPU, array programs can achieve significantly less kernel execution time. However, due to the constraint of the current GPU architecture, data transmission may need a large amount of time, slowing down the overall performance. In our experiments, we identify fusion-based optimizations work better on the CPU than on the GPU because they introduce more complex expressions that result in more data movement, and the benefit of eliminating intermediate results is less than the penalty of the data movement. Nevertheless, we are still optimistic about the integration of GPU for analytical functions, but the research direction should lie in exploring optimizations to identify expensive computations, estimate the cost of these computations on both CPU and GPU including the transfer cost, and decide which portions should be sent to the GPU.

Chapter 9
Related Work

Research in array programming languages and database systems has comprehensively studied how to improve the performance of both language processors and database query engines. In both contexts, domain knowledge has been employed in their optimizers to have a better understanding of how to optimize different codebases. The work on array programming languages has been mainly focused on array-based computation, as commonly found in compute-intensive applications, and this motivates research interests in designing, implementing, and optimizing high-performance built-in functions. Array programming is widely adopted in the fields of engineering, finance, and numeric computation. Optimization in database systems has focused on SQL queries which provide a high-level programming scheme and a rigid structure. The database community has started to embrace in-memory database systems to further improve system performance. Research in query compilers prevails because compiling database queries to efficient low-level code has become practical for in-memory database systems.

9.1 Database Query Processing

A database query can be processed in a traditional query engine with an interpreter-based environment, or a modern query compiler with a compiler-based environment. Both have been the subject of interesting prior work.

9.1.1 Traditional Query Engines

Traditional database systems have interpreter-based query engines following the design of the iterator model [37]. Executing an execution tree from leaves to the root, a node retrieves records from its children nodes as needed. This avoids generating large intermediate results and helps reduce memory footprint because of the small number of tuples processed at a time. This model works well when the I/O cost is expensive and dominates the overall performance of the query because I/O processing can overlap with processing records that are already fetched. Typical examples are the popular database systems PostgreSQL [80] and MySQL [63].

However, these traditional execution engines do not offer optimal plans anymore when most of the data can reside in memory. Thus, a new generation of database systems has been developed with the capability of storing the data entirely in memory with the purpose of removing the expensive overhead in I/O to improve query performance [35, 38, 70]. Well-known examples are the database systems SAP HANA [32] and VoltDB [87].

9.1.2 Modern Query Compilers

Compiling an SQL query has become more common, and typically means compiling SQL queries to an intermediate representation before generating machine code, rather than compiling to machine code directly. A query compiler usually prefers a data-centric model that employs operator fusion when generating efficient target code.

HyPer [64] a column-based RDBMS, aims at improving the query performance by compiling SQL to LLVM, exploiting LLVM's compiler optimization mechanisms. It first compiles an SQL query to an optimized execution plan which contains algebraic expressions. When compiling expressions of the tree to an imperative program, HyPer follows a data-centric model to generate query execution code. It focuses on the data rather than the operator and defines a *pipeline breaker* to occur when the data is moved out of CPU registers. The data usually is pushed from one pipeline breaker to another pipeline breaker. Furthermore, it introduces the *produce/consume* schema for code generation: the produce function pushes the result of the operator towards

the consume function. Thus, this scheme allows the generated code to have fewer intermediate results. This is akin to our fusion-based optimizations that fuses built-in functions. Additionally, HyPer's optimizer also offers a wide range of database-centric optimizations between relational algebra operators, such as merging adjacent joins. Similar to HyPer, MemSQL [22], a commercial database system, provides a fast SQL compiler for LLVM-based code generation. We use HyPer's execution plans as input to our HorseSQL translator as outlined in Section 5.1. In terms of optimizations, HorsePower is less database centric, focusing mainly on loop fusion, but also enables source database patterns for fusion.

Other than LLVM code, there are multiple choices for the generated code. DBToaster [12] targets high-performance delta processing in data streams by compiling SQL to C++ code. LegoBase [48] first compiles execution plans to Scala code, then compiles the Scala code to C code using the Light Modular Staging (LMS) compiler, and finally compiles the C code to the binary code. DBLAB [77, 81] proposes multiple IRs for implementing an efficient query compiler using a high-level programming language. These IRs are various Domain-Specific Languages (DSL) developed on Scala. The input execution plan is compiled through these IRs with specific domain-specific optimizations in each IR before generating the target C code. The generated C code is further compiled by a traditional C compiler. A different choice for the generated code is Java bytecode which can be later optimized in the Java Virtual Machine with the support of just-in-time compilers [74]. These systems rely on a generic compiler for generating optimized target code. Even though these compilers excel at optimizing procedural code, they know little about what a query does at a conceptual high-level. HorseIR can optimize queries with a relatively high-level view by representing queries as array-based programs with less code but with more information about the queries themselves that it then exploits for compiler optimizations.

Voodoo [69] is a declarative intermediate algebra designed with a set of vector operators. These operators are similar to HorseIR built-in functions, such as arithmetic and comparison operations, but with a lower-level design having more control over operators. An SQL query is first compiled to Voodoo code in a database, for

example, MonetDB [40]. Then, the Voodoo code is compiled to efficient parallel GPU code. HorseIR plays a similar role as Voodoo in representing code generated from database queries with vector-based data structures. Additionally, HorseIR can handle code from other source programming languages, not only SQL. HorseQC [34] is an experimental system for compiling database queries to the GPU. With a different hardware layout on the GPU, it uses a compiler approach to process entire vectors for database operators instead of the traditional tuple-at-a-time approach. It generates both CUDA and OpenCL code directly for different kinds of GPUs. This is different from our HorseGPU that generates a higher-level C code annotated with OpenACC which later can be compiled to various platforms.

MonetDB [40] first compiles SQL queries into its low-level IR, MAL, which provides a set of operations for the objects designed for column-based database systems. Its MAL optimizer generates optimized MAL code which is later sent to an interpreter-based engine for execution without further optimizations. HorseIR is different from MAL as it provides additional back-end support that further compiles HorseIR to efficient low-level code with sophisticated code optimizations.

9.2 Array Programming Languages

Our HorseIR is influenced by languages supporting array programming: ELI [19], an array-based language which mimics SQL queries; Q [53], an array-based query language supported in its database system KDB+; and Q'Nial [44], a language for powerful array computations. In particular, KDB+, which has been adopted in financial domains, is designed for fusing SQL and array programming languages. Its database system was implemented in the array programming language Q, which is a scripting language executing on an interpreter-based environment. It provides an SQL interface as a form of a wrapper on top of the language Q. The system internally manages the data, while seamlessly supporting array programming for data analytics. However, with an interpreter-based design, its performance heavily relies on hand optimizations rather than systematic compiler optimizations.

In terms of language performance, popular array programming languages, such as MATLAB [56] and R [82], are often considered inefficient as their code is usually executed on interpreter-based systems. Therefore, programmers need to write code intelligently and avoid inefficient code patterns [36], such as using a highly-tuned built-in function instead of an explicit for-loop. This relies on the experience of programmers and requires manual inspection that may lead to poor performance. Mc2Mc [21] provided an automatic solution by introducing a source-to-source translator which transforms inefficient loop patterns into efficient built-in functions with the support of vectorization.

With the support of compiler optimizations, the performance of an array language can be improved significantly. However, having a compiler for an array language is challenging as the array language usually lacks explicit type and shape information. Prior research in the performance improvement of array languages for data analytics lies in compiling them to efficient low-level code directly. For example, a research compiler from MATLAB to FORTRAN 90 [75] showed significant speedup of the compiled code compared to the MATLAB program. This compiler employed static analysis with type, shape, and value propagation. Furthermore, it had symbolic value inference for shape propagation to get precise shape information. A compiler for ELI to C code [18] showed that array languages can have a sophisticated compiler which can achieve bootstrapping. Bootstrapping defines that a language's compiler written in this language can compile the code base of the compiler. That is, an ELI-to-C compiler written in ELI can be compiled to a binary, which can translate the ELI program to C code. This research demonstrated that array languages are competent at complex programming tasks, including writing a compiler. These compilers can generate efficient target code directly without using any customized IR.

On the other hand, as array languages are high-level programming languages, it is common to introduce an array-based IR with the purpose of supporting various backends and exploring further optimization opportunities on the IR level. Octave [65], an open-source version of MATLAB, addressed the performance issue by providing a compiler for static type and shape inference to generate efficient code. McSAF [28]

provided a compiler toolkit for static analysis on MATLAB and scientific programming that helped programmers write program analyses easily. Tamer [30], another compiler toolkit, designed Tame IR for representing dynamic MATLAB programs and supported advanced value analysis for collecting precise information, such as MATLAB types and call graphs. A use case of Tamer is Mc2FOR [55], a compiler tool built on Tamer for compiling MATLAB to FORTRAN, which shows that even the performance of a dynamic scripting language can be improved with a compiler. We use TameIR in our HorseMATLAB translator as outlined in Section 5.2. Other than MATLAB, research in the array programming language APL explored compiler support using an IR as well. A typed IR [31] showed the usefulness of the IR to generate efficient compiled code. MIX10 [51] presented the code compilation from MATLAB to an IR before generating large-scale parallel code. Our HorsePower can handle a subset of MATLAB programs while introducing code optimizations considering both MATLAB and database queries.

HorseIR relies on many optimizations specifically developed for array programming languages to generate efficient parallel code, such as loop fusion known from array-based APL [24] and FORTRAN 90 [47]. However, focusing on code that represents SQL queries, these optimizations had to be adapted to the specific HorseIR context which goes well beyond a pure array programming language.

9.3 Data Analytics in Database Systems

When integrating data analytics into a database system, a direct approach is to extend SQL syntax to support data analytics by adding new features, such as the support of array programming. For example, SciQL [93] provides array-based extensions to SQL for scientific computing, and ArrayUDF [29] introduces a customized UDF system allowing user-defined operations on adjacent cells of an array. They use the array design to improve the performance of SQL queries and offer the possibility of mixing SQL queries and array programming. Compared to HorseIR in HorsePower, they support more general arrays targeted at specific domains, while HorseIR keeps vector

and list as primary data structures for efficient core SQL support. GraphScript [67], as a domain-specific graph query language tailed for graph analysis tasks, is built on top of the commercial database SAP HANA. It introduces specific types for graph data stored in relational tables and implements graph-specific optimizations. We provide a more general solution for data analytics in database query processing by combining database queries and array languages via HorseIR and performing fusion-based optimizations for array operations.

An alternative to extending existing database systems is to introduce a third-party system to process the data offloaded from the database system. An example is VoltDB [87], an in-memory database system that provides a subsystem that can fan out data to a third-party analytical tool. Thus, subsets of data stored in the database can be selected and then exported to be processed in a UDF outside the database. However, this introduces a performance problem due to the expensive data movement between independent systems. We introduce a different approach by integrating database queries and UDFs into the same intermediate layer to avoid such expensive data movement.

Froid [73] presents an approach which has some similarities with our HorsePower in executing SQL statements that include UDFs. They provide a holistic optimization solution for data analytics in database systems by adopting the UDF facilities provided by the Microsoft SQL Server and rewriting the UDFs with SQL statements to relational code. In order to translate UDFs using imperative statements to relational code, they explore a set of imperative statements allowed in relational algebraic expressions. For example, the relational code `SELECT CASE WHEN ... THEN ... ELSE END ...;` is able to represent the if-else statement `if(...) ... else ...;`. After the relational code is generated, it is optimized with the existing query optimizer. However, this approach is limited as not all UDFs are translatable to a relational operator. HorsePower is more flexible in this aspect, as it can also translate non-relational operations written in conventional languages such as MATLAB.

Weld [66] is a common runtime environment that allows code generation from its functional IR (WeldIR) exploiting different libraries, such as C and relational libraries. For example, the NumPy library [6] has many low-level functions written in C

that can be implemented in WeldIR, including the direct access to NumPy's internal data representation, such as NumPy arrays. WeldIR can handle various data processing tasks, including relational operators and functional APIs like Spark [92]. Its optimization strategy follows the operator fusion logic that fuses a chain of database operators to generate a single loop. Thus, optimizing the code from different kinds of source code becomes possible. However, in the code generation, WeldIR implements a rule-based optimization strategy for element-wise and common-loop-head fusion, while HorsePower employs automatic fusion along with pattern-based fusion. Even though WeldIR is a common IR for many languages, it is challenging to generate it automatically from these languages, such as generating WeldIR automatically from the lower-level programming language C. In contrast, HorsePower provides translators that can automatically generate HorseIR from database execution plans and MATLAB programs.

Lara [52] is a domain-specific language (DSL) tailored to relational algebra and UDFs that are used in the preprocessing prior to training a machine learning model. Its code is first compiled to an IR which is able to inspect UDFs by collecting necessary information from UDFs, including variable types, and read and write accesses to the data. This can be achieved in Lara because it is a quotation-based DSL. Similar to reflection in Java, Lara allows access to the entire ASTs of UDFs written in general-purpose programming languages. Thus, Lara can optimize such transparent UDFs together with its IR code. This is different from HorsePower which compiles database queries and UDFs to its common IR with holistic optimizations enabled.

Finally, the most popular approach has been to integrate an execution environment for popular analytical tools and languages inside database systems. This "black-box" approach is what MonetDB provides with an embedded Python interpreter and UDF constructs [72, 71]. However, as we saw in the evaluations, the data movement from a database system to Python is expensive due to the data copy and conversion between two different systems. Further, as also demonstrated in our evaluations, such a black-box implementation results in sub-optimal execution plans, reducing the optimization opportunities across the database system engine and the UDF execution environment. Being a unified system that is capable of translating both SQL and the

analytical languages used for UDFs into a common IR, HorseIR can overcome these hurdles, providing a holistic optimization and execution environment.

9.4 Compiler Optimizations

Fusion techniques are popular due to their success in reducing the number of loops with fewer intermediate results. This is true for compiled environments that easily allow code fusion.

In the MATLAB-to-FORTRAN compiler [75], shape and size information can be obtained from a conformability analysis with a set of well-defined conforming 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` [88]. The function `Apply` is similar to our list-based functions, which takes a function and a list of data inputs as parameters and repeatedly applies the function. Their vectorizer involves both code and data transformation while we focus on code optimizations. Additionally, we explore a wider set of built-in functions rather than only a single list-based function.

Ju et al. [45] investigate built-in function fusion in a pure array-based programming language, APL. Array-based functions are classified based on the features and present a new approach to help generate parallel code. Ching et al. [24] 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 [51, 33], we exploit the 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 [58, 21].

Loop fusion has been exploited for query execution, and has been typically achieved through a fixed set of complex pre-defined rules. The DBLAB query compiler [77] provides rules for different loop fusion algorithms to generate optimized

code [76]. HorseQC [34] needs to chain a list of operators for the fusion before sending code to be executed on the GPU. This is better than the kernel-at-a-time approach in that the transmission of the intermediate results can be avoided. HIQUE [49] is a prototype system designed for compiling database queries to generate optimized query- and hardware-specific code. Both HorseQC and HIQUE adopt a rule-based fusion strategy. However, like the patterns in HorseIR [20], rules are challenging to generalize.

Peloton [57] considers operator fusion, mixed pipelined tuple-at-a-time processing and vectorization, for operators within a pipeline. Stream-fusion [76] needs to define extra fusion-related constructs for collections. By contrast, our fusion strategy is a systematic approach that collects precise shape information on a well-defined array-based language for automatic fusion.

Other than in the domain of database query processing, there are many performance-oriented systems that adopted fusion-based optimization strategies to improve system performance. TVM [23], 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 with specific patterns. We provide a more sophisticated fusion approach that considers more groups and defines systematic data-flow analysis to identify fusion opportunities. LIFT [50] shows a different approach by defining complex functional patterns with precise descriptions for generating efficient code for parallel devices such as an FPGA, as its input code can come from various domains beyond the domain of database queries.

Chapter 10
Conclusions and Future Work

In this thesis we presented HorsePower, a compiler-driven system to efficiently execute SQL queries and data analytics programs. The main contributions of this thesis are (i) the design and implementation of HorsePower; (ii) the exploration of marrying database query processing and array programming languages; and (iii) the thorough experiments for validating the effectiveness of the system.

10.1 Conclusions

By adding the support of HorseIR, a newly designed array-based IR for database queries and array programming languages, HorsePower is capable of handling database queries as well as integrating analytic functions writing in array programming languages into database queries. HorseIR is an array-based intermediate representation tailored for database queries in column-based in-memory database systems, and array programming languages. Our system HorsePower supports effective optimizations of HorseIR programs generated from various source languages to generate efficient target code.

HorsePower provides automatic translators for generating HorseIR code from either the execution plans of database queries or MATLAB programs, or both of them. After a HorseIR program is optimized, multiple back-ends can compile the optimized HorseIR code to generate efficient target code for various platforms.

Various kinds of experiments were conducted for HorsePower. The first experiment showed that the performance of HorsePower is comparable and even superior to a state-of-the-art database system with the standard database benchmark TPC-H. With running on multiple threads and scales, the capabilities of parallelism and scalability were confirmed. The second experiment tested that HorsePower is able to manage pure array programming code and generate well-optimized code. Two array-based benchmarks written in MATLAB were compiled to HorseIR. The third experiment suite tested that database queries that have integrated user-defined functions can be handled and optimized in a holistic manner in HorsePower. Two kinds of user-defined functions were created, one derived from standard queries, the other from our MATLAB benchmarks. The results of this experiment showed that HorsePower provides an efficient system for handling and optimizing mixed code in a united way. The last benchmark scenario tested HorsePower's capability of executing both on the CPU and the GPU in a united manner. We show that execution compute-intensive code on the GPU is beneficial but data transfer between CPU and GPU can be a bottleneck.

The research in this thesis is a first step in exploring how to improve the performance of database query processing using HorseIR, which was designed to represent both database queries and general array programming programs in the context of in-memory column-based database systems. HorsePower, as a system, provides necessary components to complete the translation from source languages to target C code generation via HorseIR. Thus, the performance of database query processing can be improved using compiler optimization techniques. In addition, our experiment results have shown the potential to support the applications which need high performance in data analytics along with database queries.

10.2 Future Work

HorsePower is an extensible and growing system. We believe there are many possible future research directions that could be followed. We outline some of them here.

- One research direction is to enable a customized system for domain-specific purposes. Other than an optimizing system for database queries and array programming languages, different contexts can be supported by adding more data types and built-in functions to support applications from new domains, such as scientific computation and machine learning. This brings additional challenges in providing a general-purpose optimization strategy for optimizing array-based programs precisely.

- HorsePower relies on optimized execution plans for database queries and on MATLAB to provide already optimized TameIR code. To be independent from such work and also to further support other languages where such "pre-optimizations" are unavailable, HorsePower could be extended to provide more commonly used dataflow analyses, such as constant propagation and common sub-expression elimination.

- Research in offering new back-ends is encouraging because of the recent emergence of new parallel hardware for compute-intensive programs. Since many of these hardware designs follow the scheme of data parallelism, legacy code from database queries and array programming languages can be deployed to such hardware swiftly and efficiently.

- HorsePower so far focuses on query processing. But it has already the basic components to also support other database functionality. Still, significant extensions are needed to support the first phase of query optimizations, i.e., execution plan generation, as well as offering more sophisticated code management (data definition and transaction management).

Appendix A
HorseIR Language Specification

In this appendix we introduce the HorseIR language specification in Appendix A.1, the value ranges of HorseIR numeric types in Appendix A.2, and brief introduction of HorseIR built-in functions in Appendix A.3.

A.1 Language Grammar

```
letter           = 'a' ... 'z' | 'A' ... 'Z' ;
digit            = '0' ... '9' ;
nzdigit          = '1' ... '9' ;
digits           = digit { digit } ;
ascii_character  = /* All valid ASCII */ ;
escape_sequence  = "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v" ;
Identifier       = ( letter | '_' ) { letter | digit | '_' } ;

Values        = ValueList ':' Type
ValueList     = Value | '(' Value { ',' Value } ')'
Value         = IntValue  | FloatValue  | BoolValue   | ComplexValue |
                CharValue | StringValue | SymbolValue | CalendarValue
   ;

Sign          = '+' | '-'
Integer       = '0' | nzdigit { digit } ;
```

```
16 Float         = Integer '.' [ digits ] | '.' digits ;
17
18 IntValue      = [ Sign ] Integer ;
19 FloatValue    = [ Sign ] Float ;
20 BoolValue     = '0' | '1' ;
21
22 CharValue     = "'" ascii_character "'" ;
23 StringValue   = '"' { ascii_character } '"' ;
24 SymbolValue   = '`' ( Identifier | StringValue ) ;
25 ComplexValue  = FloatValue [ Sign Float ] 'i' ;
26
27 CalendarValue = DateTimeValue | MonthValue  | DateValue |
28                 MinuteValue   | SecondValue | TimeValue ;
29
30 // Must correspond to valid dates within range, see the type
       declarations
31 MonthValue    = Integer '-' Integer ;
32 DateValue     = Integer '-' Integer '-' Integer ;
33 MinuteValue   = Integer ':' Integer ;
34 SecondValue   = Integer ':' Integer ':' Integer ;
35 TimeValue     = Integer ':' Integer ':' Integer '.' Integer ;
36 DateTimeValue = DateValue 'T' TimeValue ;
37
38 // Module
39 Module          = "module" Identifier '{' ModuleContents '}' ;
40 ModuleContents  = { ImportDirective | FunctionDeclaration |
       GlobalDeclaration } ;
41
42 // Import directives
43 ImportDirective  = "import" Identifier '.' ImportList ';' ;
44 ImportList = '*' | Identifier | '{' Identifier { ',' Identifier } '}'
       ;
45
46 // Function declarations
47 FunctionDeclaration  = FunctionKind Identifier '(' Parameters ')'
48                          [ ':' ReturnTypes ] Block ;
49
```

```
50 FunctionKind  = "def" | "kernel" ;
51
52 Parameters    = [ Parameter { ',' Parameter } ] ;
53 Parameter     = Identifier ':' Type ;
54
55 ReturnTypes   = Type { ',' Type } ;
56
57 Type       = Wildcard | BasicTypes | ListType |
58              DictType | EnumType | TableTypes ;
59 Wildcard   = '?' ;
60 BasicTypes = "bool" | "i8"   | "i16"     | "i32"    | "i64"    |
61              "f32"  | "f64"  | "complex" | "char"   | "str"    |
62              "sym"  | "dt"   | "date"    | "month"  | "minute" |
63              "second" | "time" ;
64 ListType   = "list" '<' Type { ',' Type } '>' ;
65 DictType   = "dict" '<' Type ',' Type '>' ;
66 EnumType   = "enum" '<' Type '>' ;
67 TableTypes = "table" | "ktable" ;
68
69 Block        = '{' { Statement } '}'
70 ControlBlock = Statement | Block
71 Statement    = AssignStmt | ControlStmt | ExpressionStmt | VarDecl ;
72
73 AssignStmt   = VarList '=' Expression ';' ;
74 VarList      = Var { ',' Var } ;
75 Var          = Identifier [ ':' Type ] | Identifier '.' Identifier ;
76
77 ControlStmt  = IfStmt    | WhileStmt | RepeatStmt | ReturnStmt |
78                BreakStmt | ContinueStmt ;
79 Condition    = Operand ;
80
81 IfStmt       = "if" '(' Condition ')' ControlBlock [ "else"
      ControlBlock ]
82 WhileStmt    = "while" '(' Condition ')' ControlBlock ;
83 RepeatStmt   = "repeat" '(' Condition ')' ControlBlock ;
84 ReturnStmt   = "return" [ { Operand { ',' Operand } ] ';' ;
85 BreakStmt    = "break" ';' ;
```

```
86 ContinueStmt = "continue" ';' ;
87 ExpressionStmt = Expression ';'
88 VarDecl = "var" Identifier [ { ',' Identifier } ] ':' Type ';' ;
89
90
91 // Expressions
92 Expression   = FunctionCall | Operand | Cast ;
93 FunctionCall = FunctionId '(' [ Operands ] ')' ';'
94 FunctionId   = '@' Identifier [ '.' Identifier  ] ;
95 Operands     = Operand { ',' Operand } ;
96 Operand      = Identifier [ '.' Identifier ] | Literal ;
97 Literal      = FunctionLiteral | VectorLiteral ;
98
99 FunctionLiteral  = FunctionId [ ':' "func" ] ;
100 VectorLiteral    = Value ':' Type | '(' Value { ',' Value } ')' ':'
    Type ;
101 Cast = "check_cast" '(' Expression ',' Type ')' ;
```

A.2 Value Ranges

Numeric types have value ranges following standard C conventions and all are signed types.

boolean (bool)

0 (False) or 1 (True)

small (i8)

-2^7 to $2^7 - 1$

short (i16)

-2^{15} to $2^{15} - 1$

int (i32)

-2^{31} to $2^{31} - 1$

long (i64)

-2^{63} to $2^{63} - 1$

float (f32)

$1.2e^{-38}$ to $3.4e^{+63}$

double (f64)

$2.3e^{-308}$ to $2.3e^{+308}$

complex (complex)

consists of two floating point numbers (f32)

date - year (YYYY)

1000 to 9999

date - month (MM)

01 to 12 (two digits required)

date - day (DD)

01 to 28/29/30/31 (depends on month and year)

- January - 31 days
- February - 28 days (common year) and 29 days (leap year)
- March - 31 days
- April - 30 days
- May - 31 days
- June - 30 days
- July - 31 days
- August - 31 days
- September - 30 days
- October - 31 days
- November - 30 days

- December - 31 days

date - hour (hh)

00 to 23

date - minute (mm)

00 to 59

date - second (ss)

00 to 59

date - millisecond (lll)

000 to 999

A.3 Built-in Functions

1. Unary Built-in functions (Element-wise)

abs(x), neg(x), conj(x)

One takes a numeric parameter and returns its absolute, negated, or conjugate value.

ceil(x), floor(x), round(x)

One takes a numeric parameter and returns its ceiling, flooring, or rounding value.

recip(x)

One takes a numeric parameter and returns its reciprocal value (i.e., 1/x).

sigsum(x)

One takes a numeric parameter and returns {0 if x==0, -1 if x<0, 1 if x>0}.

pi(x)

One takes a numeric parameter and returns (`x * ` π).

not(x)

One takes a boolean parameter and returns its negated value.

log(x), log2(x), log10(x)

One takes a real parameter and returns its log value as $ln(x)$, $log_2(x)$, and $log_{10}(x)$.

sqrt(x), exp(x)

One takes a real parameter and returns its sqrt ($\sqrt{x}$) and exponential (e^x) value.

date(x), date_year(x), date_month(x), date_day(x)

One takes a date parameter and returns its partial value.

time(x), time_year(x), time_month(x), time_day(x)

One takes a time parameter and returns its partial value.

sin(x), cos(x), tan(x), asin(x), acos(x), atan(x)

One takes a numeric parameter, computes with a trigonometric function, and returns its result.

sinh(x), cosh(x), tanh(x), asinh(x), acosh(x), atanh(x)

One takes a numeric parameter, computes with a trigonometric function, and returns its result.

2. Unary Built-in functions

unique(x)

One returns the unique items in the input data.

len(x)

One returns the length of the input data.

range(x)

One returns a list of consecutive numbers from 0 to x-1, inclusive.

fact(x)

One returns the factorial of the input number.

rand(x)

One takes an integer x as an range and returns a random integer from 0 (inclusive) to x (exclusive).

seed(x)

One takes an integer and set it as a global seed. (Default: 16807)

where(x)

One returns the indices of elements where their values are true.

group(x)

One groups the input data and returns the indices of grouped items.

sum(x), avg(x), min(x), max(x)

One takes a numeric parameter and returns its total (sum), average (avg), minimum (min), or maximum (max) value.

3. Binary built-in functions (Element-wise)

lt(x,y), gt(x,y), leq(x,y), geq(x,y), eq(x,y), neq(x,y)

Comparison functions for less than (lt), greater than (gt), less than equal (leq), greater than equal (geq), equal (eq), and not equal (neq).

plus(x,y), minus(x,y), mul(x,y), div(x,y)

Arithmetic functions for addition (plus), minus (minus), multiplication (mul), and division (div).

power(x,y), logb(x,y), mod(x,y)

Math functions for power (x^y), logarithm ($log_x(y)$) and modulo (y mod x).

and(x,y), or(x,y), nand(x,y), nor(x,y), xor(x,y)

Boolean functions for logical and (and), or (or), nand (nand), nor (nor), and xor (xor).

4. Binary built-in functions

append(x,y)

One appends all items in y to x and returns a new vector or list.

like(x,y)

One finds all matched items in x based on the pattern y and returns a boolean vector.

compress(x,y)

One takes two same-length vectors and selects items from y based if the corresponding positions in the boolean vector x are true.

index_of(x,y)

One finds each item from x in y, and returns the position if found, otherwise the length of y.

order(x,y)

One sorts x based on the orders in y and returns the indices after sorting.

member(x,y)

One finds items from x in y ($x \in y$) and returns a boolean vector indicating found (True) or not (False).

vector(x,y)

One initializes a vector by replicating the value y with x times.

5. List-based built-in functions

list(...)

One returns a list containing an arbitrary number of arguments as its cells.

tolist(x)

One takes a parameter and returns a one-cell list in which the cell is the parameter.

raze(x)

One returns a vector containing the value in each cell in the input list x.

each(fn, x)

One applies the function `fn` on the each cell of x and returns a new list.

each_item(fn,x,y), each_left(fn,x,y), each_right(fn,x,y)

One applies the function `fn` on each cell (x[i],y[i]), left cell only (x[i],y), or right cell only (x,y[i]), and returns a new list.

6. Database-related built-in functions

enum(x,y), dict(x,y), table(x,y), ktable(x,y)

One returns an enumeration (enum), dictionary (dict), table (table), keyed table (ktable), based on input data.

keys(x), values(x)

One fetches the information of keys (keys), values (values) for enumeration, dictionary, table, or keyed table.

keys(x), values(x), meta(x)

One returns the meta information for table.

load_table(x)

One loads a table by a given name.

column_value(x,y)

One loads a column from x by a given name y.

fetch(x)

One fetches the original value from an enumeration.

join_index(op,x,y)

One takes join operators, a list of vectors as the left side of the join, and another list of vectors as the right side of the join, and returns a list of two vectors indicating the indices of joined items from the left and right sides.

7. Miscellaneous built-in functions

index(x,y)

One fetches values from x by given indices in y.

index_a(x,y,v)

One updates the vector x with the given indices in y and the new values in v.

sub_string(x,a,n)

One returns the substrings of strings in x with a given range $[a, a + n)$.

print(x)

One prints the value x.

Appendix B
Plan-to-HorseIR Translator Specification

In order to have a better understanding of the design and implementation of the translator for generating HorseIR from database plans, we create a specification of the plan-to-HorseIR translator introduced in HorseSQL Section 5.1. The translator takes a HyPer's plan in the JSON format as input, and emits a HorseIR program as output. As the input is a JSON object rather than a stream of tokens, we simplified key and value pairs in JSON by using the key as a leading literal string token, and the value as a terminal or non-terminal. Then, we can have the specification described using the extended Backus-Naur form (EBNF) with the following list of useful notations:

- `{…}` for repeating terminals or non-terminals,
- `[…]` for optionally selecting terminals or non-terminals,
- `|` for choosing one from terminals or non-terminals, and
- `"…"` for a literal string token.

The details of the specification can be found as follows:

```
1 /*
2  * Basic terminals:
3  *   Integer, String, StringList, *Id
4  */
5 Plan          = PlanHeader PlanBody ;
```

```
6  PlanHeader    = "header" { NameId AliasId } ;
7  PlanBody      = "plan" Input ;
8  PlanCommon    = "operatorId" Integer [ "cardinality" Integer ] ;
9  Input         = PlanCommon SubInput ;
10 SubInput      = "operator" "tablescan"     ScanTableScan
11               | "operator" "tempscan"      ScanTempScan
12               | "operator" "groupbyscan"   ScanGroupbyScan
13               | "operator" "groupby"       ScanGroupby       "input"
       Input
14               | "operator" "sort"          ScanSort          "input"
       Input
15               | "operator" "map"           ScanMap           "input"
       Input
16               | "operator" "select"        ScanSelect        "input"
       Input
17               | "operator" "earlyprobe"    ScanEarlyProbe    "input"
       Input
18               | "operator" "temp"                            "input"
       Input
19               | "operator" "join"          ScanJoin          "left"
       Input "right" Input
20               | "operator" "groupjoin"     ScanGroupJoin     "left"
       Input "right" Input
21               | "operator" "leftantijoin"  ScanLeftAntiJoin  "left"
       Input "right" Input
22               | "operator" "rightantijoin" ScanRightAntiJoin "left"
       Input "right" Input
23               | "operator" "leftsemijoin"  ScanLeftSemiJoin  "left"
       Input "right" Input
24               | "operator" "rightsemijoin" ScanRightSemiJoin "left"
       Input "right" Input ;
25 ScanGroupby   = "values" Values "aggregates" Aggregates ;
26 ScanSelect    = "condition" Condition ;
27 ScanTableScan = "segment" Integer "from" TableId \
28                 "values" Values "tid" Iu "tableOid" IuSpecial \
29                 "tupleFlags" StringList "restrictions" Restrictions \
30                 [ "residuals" Residuals ] ;
```

```
31 ScanSort      = "criterion" Criterion [ "count" Count ] ;
32 ScanMap       = "values" Values ;
33 /* scans */
34 ScanJoin         = "method" Method \
35                    "singleMatch" SingleMatch \
36                    "condition" Condition [ "magic" Magic ] ;
37 ScanLeftSemiJoin = ScanJoin ;
38 ScanRightSemiJoin= ScanJoin ;
39 ScanLeftAntiJoin = ScanJoin ;
40 ScanRightAntiJoin= ScanJoin ;
41 ScanGroupJoin    = "leftKey" expressions "rightKey" Expressions "
       compareTypes" Types \
42                    "leftExpressions" Expressions "rightExpressions"
       Expressions \
43                    "leftCollates" StringList "rightCollates"
       StringList \
44                    "leftAggregates" Aggregates "rightAggregates"
       Aggregates \
45                    "semantic" "outer" ;
46 CommonScan       = "source" Integer "output" Output ;
47 ScanTempScan     = CommonScan ;
48 ScanGroupbyScan  = CommonScan ;
49 ScanLeftOuterJoin= "condition" Condition "magic" Magic ;
50 ScanLeftMarkJoin = "condition" Condition ;
51 ScanEarlyProbe   = "values" StringList "builder" Integer ;
52 /* expressions */
53 Values        = { Value } ;
54 Value         = Expressions
55               | "iu" Iu "name" TargetId
56               | "iu" Iu "value" Expressions ;
57 Iu            = Id [ type ] ;
58 IuSpecial     = Id "RegClass" ;
59 Aggregates    = { AggrItem } ;
60 AggrItem      = "source" Integer "operation" Operation "iu" Iu ;
61 Restrictions  = { RestrictCell } ;
62 Restrict_cell = "attribute" Integer "mode" Mode "value" Value [ "
       value2" Value ] ;
```

```
63 Expressions   = "expression" "comparison" "mode" Mode "left"
       Expressions "right" Expressions
64               | "expression" "quantor"    "mode" "=some" "arguments" {
        Expressions }
65               | "expression" ExprArgn    "arguments" { Expressions }
66               | "expression" ExprArg2    "left" Expressions "right"
       Expressions
67               | "expression" "const"      "value" ConstValue
68               | "expression" "lookup"     "input" VarValue "values"
       ConstValue
69               | "expression" "isnotnull"  "input" VarValue
70               | VarValue ;
71 ConstValue    = "type" Type "value" StringValue ;
72 VarValue      = "expression" "iuref" "iu" Iu ;
73 Criterion     = "nullFirst" isNullFirst "descending" isdesc "value"
       Value ;
74 Residuals     = { Expressions } ;
75 Condition     = Expressions ;
76 Magic         = Input ;
77 Output        = { SourceTarget } ;
78 SourceTarget  = "source" Id "target" Iu ;
79 /* constants */
80 Types         = { Type } ;
81 Type          = TypeBasic [ "nullable" ] ;
82 TypeBasic     = "Char1"
83               | "Char" Integer
84               | "BigInt"
85               | "Varchar" [ Integer ]
86               | "Integer"
87               | "Date"
88               | "Numeric" Integer Integer
89               | "Bool" ;
90 Operation     = "keep"
91               | "count"
92               | "sum"
93               | "avg"
94               | "min"
```

```
95                | "max"
96                | "any"
97                | "countdistinct" ;
98  Method        = "indexnl"
99                | "hash"
100               | "bnl" ;
101 Mode          = "="
102               | "[)"
103               | "[]"
104               | "(]"
105               | ">"
106               | "<"
107               | "<>"
108               | "=some"
109               | ">="
110               | "<=" ;
111 ExprArgn      = "and"
112               | "or"
113               | "like"
114               | "substring"
115               | "between" ;
116 ExprArg2      = "mul"
117               | "sub"
118               | "div" ;
119 /* Basics */
120 letter        = 'a' ... 'z' | 'A' ... 'Z' ;
121 digit         = '0' ... '9' ;
122 nzdigit       = '1' ... '9' ;
123 digits        = digit { digit } ;
124 ascii_character  = /* All valid ASCII */ ;
125 Id            = ( letter | '_' ) { letter | digit | '_' } ;
126 NameId        = Id ;
127 AliasId       = Id ;
128 TableId       = Id ;
129 TargetId      = Id ;
130 Integer       = '0' | ('+' | '-') nzdigit {digit} ;
131 StringValue   = '"' { ascii_character } '"' ;
```

```
132 StringList    = '[' [ StringValue { ',' StringValue } ] ']' ;
```

Bibliography

[1] Flex: Lexical Analyser Generator. https://github.com/westes/flex. [Last accessed in October 2020].

[2] GCC, the GNU Compiler Collection. https://gcc.gnu.org. [Last accessed in October 2020].

[3] GNU Bison: A General-purpose Parser Generator. https://www.gnu.org/software/bison. [Last accessed in October 2020].

[4] HorseIR Online Documentation. http://www.sable.mcgill.ca/~hanfeng.c/horse/docs/horseir/functions/. [Last accessed in October 2020].

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

[6] NumPy Project. https://numpy.org. [Last accessed in October 2020].

[7] NVIDIA PGI Compilers. https://www.pgroup.com. [Last accessed in October 2020].

[8] OpenACC: Open Accelerators. https://www.openacc.org. [Last accessed in October 2020].

[9] OpenMP: Open Multi-Processing. https://www.openmp.org. [Last accessed in October 2020].

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

[11] The LLVM Compiler Infrastructure. http://llvm.org. [Last accessed in October 2020].

[12] Y. Ahmad and C. Koch. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *PVLDB*, 2(2):1566–1569, 2009.

[13] Austin Appleby. MurMur3 Hash. https://github.com/aappleby/smhasher. [Last accessed in October 2020].

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[15] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 54–65. Morgan Kaufmann, 1999.

[16] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, Online Proceedings, (CIDR'05)*, pages 225–237, 2005.

[17] S. Ceri and G. Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985.

[18] H. Chen, W. Ching, and L. J. Hendren. An ELI-to-C Compiler: Design, Implementation, and Performance. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, (ARRAY@PLDI'17)*, pages 9–16, 2017.

[19] H. Chen and W.-M. Ching. ELI: a simple system for array programming. *Vector, the Journal of the British APL Association*, 26(1):94–103, 2013.

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

[21] H. Chen, A. Krolik, E. Lavoie, and L. J. Hendren. Automatic Vectorization for MATLAB. In *Languages and Compilers for Parallel Computing, (LCPC'16)*, pages 171–187, 2016.

[22] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *PVLDB'16*, 9(13):1401–1412, 2016.

[23] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'18)*, pages 578–594, 2018.

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

[25] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[26] J. Davidson and A. Holler. Subprogram Inlining: A Study of Its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, 18(2):89–102, 1992.

[27] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation, (OSDI'04)*, pages 137–150, 2004.

[28] J. Doherty and L. J. Hendren. McSAF: A Static Analysis Framework for MATLAB. In *Proceedings of the 26th European Conference on Object-Oriented Programming, (ECOOP'12)*, pages 132–155, 2012.

[29] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu. ArrayUDF: User-Defined Scientific Data Analysis on Arrays. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, (HPDC'17)*, pages 53–64, 2017.

[30] A. W. Dubrau and L. J. Hendren. Taming MATLAB. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'12*, pages 503–522, 2012.

[31] M. Elsman and M. Dybdal. Compiling a Subset of APL Into a Typed Intermediate Language. In *Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, (ARRAY'14)*, pages 101–106, 2014.

[32] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Record*, 40(4):45–51, 2011.

[33] V. Foley-Bourgon and L. J. Hendren. Efficiently Implementing the Copy Semantics of MATLAB's Arrays in JavaScript. In *Dynamic Languages Symposium, (DLS'16)*, pages 72–83, 2016.

[34] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data, (SIGMOD'18)*, pages 1603–1618, 2018.

[35] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[36] P. Getreuer. Writing Fast MATLAB Code, 2006.

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

[38] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King. *Gong Show Presentation at CIDR*, pages 231–242, 2007.

[39] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

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

[41] Intel. Introduction to Intel® Advanced Vector Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions, 2011.

[42] Y. E. Ioannidis. Query Optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[43] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

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

[45] D. Ju, C. Wu, and P. R. Carini. The Classification, Fusion, and Parallelization of Array Language Primitives. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1113–1120, 1994.

[46] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the*

27th International Conference on Data Engineering, (ICDE'11), pages 195–206, 2011.

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

[48] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB'14*, 7(10):853–864, 2014.

[49] K. Krikellas, S. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *Proceedings of the 26th International Conference on Data Engineering, ICDE'10*, pages 613–624, 2010.

[50] M. Kristien, B. Bodin, M. Steuwer, and C. Dubach. High-level Synthesis of Functional Patterns with LIFT. In *ARRAY@PLDI'19*, pages 35–45, 2019.

[51] V. Kumar and L. J. Hendren. MIX10: compiling MATLAB to X10 for high performance. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, pages 617–636, 2014.

[52] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB'19*, 12(11):1553–1567, 2019.

[53] kx. KDB+ Database System. https://www.kx.com/. [Last accessed in October 2020].

[54] J. Lajus and H. Mühleisen. Efficient Data Management and Statistics with Zero-Copy Integration. In *Conference on Scientific and Statistical Database Management, (SSDBM'14)*, pages 12:1–12:10, 2014.

[55] X. Li and L. J. Hendren. Mc2FOR: A Tool for Automatically Translating MATLAB to FORTRAN 95. In *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, (CSMR-WCRE'14)*, pages 234–243, 2014.

[56] MathWorks. MATLAB. https://www.mathworks.com. [Last accessed in October 2020].

[57] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB'17*, 11(1):1–13, 2017.

[58] V. Menon and K. Pingali. A Case for Source-Level Transformations in MATLAB. In *Domain-specific Languages, (DSL'99)*, pages 53–65, 1999.

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

[60] MonetDB. MonetDB Optimizer Pipelines. https://www.monetdb.com/Documentation/SQLReference/PerformanceOptimization/OptimizerPipelines. [Last accessed in October 2020].

[61] S. S. Muchnick. Advanced Compiler Design and Implementation. chapter 4: Intermediate Representations, pages 67–104. Morgan Kaufmann, 1997.

[62] S. S. Muchnick. Advanced Compiler Design and Implementation. chapter 8.10: Du-Chains, Ud-Chains, and Webs, pages 251–252. Morgan Kaufmann, 1997.

[63] MySQL. MySQL Database. https://www.mysql.com. [Last accessed in October 2020].

[64] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB'11*, 4(9):539–550, 2011.

[65] K. Olmos and E. Visser. Turning Dynamic Typing into Static Typing by Program Specialization in a Compiler Front-end for Octave. In *The 3rd IEEE International Workshop on Source Code Analysis and Manipulation, (SCAM'03)*, pages 141–150, 2003.

[66] S. Palkar, J. J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. P. Amarasinghe, S. Madden, and

M. Zaharia. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB'18*, 11(9):1002–1015, 2018.

[67] M. Paradies, C. Kinder, J. Bross, T. Fischer, R. Kasperovics, and H. Gildhoff. GraphScript: Implementing Complex Graph Algorithms in SAP HANA. In *Proceedings of The 16th International Symposium on Database Programming Languages, (DBPL'17)*, pages 13:1–13:4, 2017.

[68] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[69] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB'16*, 9(14):1707–1718, 2016.

[70] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, (SIGMOD'09)*, pages 1–2, 2009.

[71] M. Raasveldt. Integrating Analytics with Relational Databases. In *Proceedings of the VLDB 2018 PhD Workshop co-located with the 44th International Conference on Very Large Databases, (VLDB'18))*, 2018.

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

[73] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. A. Galindo-Legaria, and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB'17*, 11(4):432–444, 2017.

[74] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE'06*, page 23, 2006.

[75] L. D. Rose and D. A. Padua. Techniques for the Translation of MATLAB Programs into Fortran 90. *ACM Transactions on Programming Languages and Systems, (TOPLAS'99)*, 21(2):286–323, 1999.

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

[77] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data, (SIGMOD'16)*, pages 1907–1922, 2016.

[78] S. Singhai and K. S. McKinley. A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.

[79] J. M. Smith and P. Y. Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Communications of the ACM*, 18(10):568–579, 1975.

[80] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[81] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, pages 307–322, 2018.

[82] R. C. Team. R: A Language and Environment for Statistical Computing. http://www.R-project.org/, 2014.

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

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

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

[86] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, (CASCON'99)*, page 13, 1999.

[87] VoltDB. VoltDB Technical Overview. https://www.voltdb.com/wp-content/uploads/2017/03/hv-white-paper-voltdb-technical-overview.pdf, 2016.

[88] H. Wang, D. A. Padua, and P. Wu. Vectorization of Apply to Reduce Interpretation Overhead of R. In *OOPSLA'15*, pages 400–415, 2015.

[89] M. Wolfe. Optimizing Supercompilers for Supercomputers. Research monographs in parallel and distributed computing, chapter 5: Loop Fusion and Loop Scalarization, pages 89–96. Pitman, 1989.

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

[91] M. Zagha and G. E. Blelloch. Radix Sort for Vector Multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 712–721, 1991.

[92] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, (HotCloud'10)*, volume 10, page 95, 2010.

[93] Y. Zhang, M. L. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, (SIGMOD'13)*, pages 1049–1052, 2013.