HorseIR : An Array-based Approach to SQL Queries

Hongji Chen
McGill University
Montreal, Quebec, Canada
hongji.chen@mail.mcgill.ca

1 Problem and Motivation

Modern software services, such as financial analysis, operate on a vast amount of data. To address the challenge of querying data, the design of the database systems and query languages has continuously evolved during the past decade. Database systems, such as MonetDB[1], utilize an IR-based analysis and optimization techniques to meet performance challenges and introduce user-defined functions (UDFs) to extend the flexibility of the SQL standard. However, the conventional IR design hides the details of database querying steps (such as joining) or separates the IR from the user-defined functions. These drawbacks significantly limit the context information propagated to the back-end and render the optimizer less effective.

In this project, we purpose an array-based IR, named HorseIR, which explores the details of query fully and connects to user-defined functions transparently. By replacing SQL and user-defined functions with built-in array-based primitives, HorseIR will enable the back-end optimizer to employ techniques developed in the programming language research field such as data-flow analysis, in the hope of handling more flexible queries efficiently.

2 Background and Related Work

MonetDB is one of the pioneers in IR based execution engine design [5]. Upon receiving a SQL query, its front-end compiles the query into a sequence of intermediate representation codes called MonetDB Assembly Language (MAL), with each instruction mapping to a built-in relational algebra primitive operation. The execution kernel later interprets the instructions step by step and generates the query result.

In 1993, Arthur Whitney launched the K programming language [12] , as a foundation for KDB system. After ten years of development, K programming language evolves into Q programming language [3] which provides a more readable interface. Both languages adopt the idea of array programming; hence they can be easily parallelized using SIMD or MIMD during execution [7].

3 Approach and Uniqueness

The state of the art architecture of MonetDB kernel inspired our design of HorseIR. Similar to MonetDB, we introduce an intermediate representation, to handle the complexity of SQL query interpretation. However, unlike MonetDB, the syntax and primitives follow the array programming style. HorseIR is a static single assignment (SSA), statically-typed, array programming language. It implements most of the primitives in array programming and database-specific data types introduced in K and Q programming languages. Compared with the traditional approaches to SQL queries, HorseIR has the following advantages:

Array-based IR with a Rich Set of Built-in Primitives In MonetDB, each primitive captures the operation in relational algebra. Although this simplifies the design of the execution engine, it hides the details from the optimizer. Thus in HorseIR, we expose such information to the back-end by introducing an array-based IR. The primitives in HorseIR are basic array-based arithmetic (e.g. factorial) or database-specific operations (e.g. table loading). Thus we can replace each relational operation with one or more array-based primitives.

Programming Language Optimization for Database In the traditional approach, the optimizations for database queries focused on relational algebra and execution plan level. Thus, in MonetDB and many other database systems, minimal optimizations are applied on the intermediate representation level. By introducing array-based HorseIR, we enable techniques developed in programming language research field to be applied to database query processing.

Extensible Framework for User-defined Functions (UDF) There is no specification for user-defined functions in SQL standard. However, to improve the flexibility of SQL queries, many database systems have introduced UDF as a language extension. However, the implementation and optimization for UDF introduce complexity in system design. The traditional approach to handling UDF is to either reduce the language flexibility (used in MonetDB, all UDF are implemented using hand-written MAL [8]) or require external linkage [10]. Our solution is to compile the UDF written in an arbitrary language into HorseIR and later compile the generated intermediate code with the SQL queries. HorseIR allows flexible UDF implementation and efficient code generation. Further, we can apply compiler optimization techniques to optimize the UDF, such as function inlining.

Efficient Parallel Code Generation Recent research attempted to introduce parallel computation into database queries using both CPU[11, 13] and GPU [4] parallelism. These research shows that the performance of database query is improved significantly using the parallel physical architecture. As HorseIR is an array-based programming language, it is easy to parallelize array-based primitives [6].

4 Results and Contributions

Language Design

Hanfeng Chen (Ph.D. mentor) sketched the general structure and back-end built-in primitives of HorseIR. I refined the IR and defined the type system. The type system allows static type checking and function overloading. These features enhance the performance and the flexibility of HorseIR. The type system classifies the variables into four unique classes: primitives, dictionaries, lists, and enumerations. Similar to MonetDB, which uses "any" keyword to declare polymorphic user-defined functions [9], HorseIR system allows polymorphic built-in and user-defined function declarations by introducing a particular data type called the wildcard type. Multiple primitive built-in function implementations are allowed in HorseIR. The overloading provided by the type system allows the interpreter to select the most efficient implementation. Unlike MonetDB which defers the type checking and function dispatching to runtime, HorseIR will resolve these during compile time, to minimize runtime overhead and improve the performance.

Front-end and Interpreter Implementation

During the summer, I have designed and implemented a complete front-end for HorseIR. The front-end provides a lexer and a parser for HorseIR. Both the lexer and parser are implemented using ANTLR4[2] , which provides an user-friendly interface generating robust LL parser. After the parsing, the front-end transforms the parse tree into a customized abstract syntax tree. The abstract syntax tree later serves as a foundation for program analysis and code transformation. From the abstract syntax tree, the front-end can generate a control-flow graph and a static call graph. An execution engine can later use this information to interpret the program efficiently. The last component of the front-end is the interpreter. Connected to Hanfeng's backend, the interpreter traverses across the abstract syntax tree, and interpret statements individually.

5 Conclusion and Future Work

In this project, we purpose a new intermediate representation for SQL database queries. The IR provides the queries with a high-performance implementation and flexible extension. The new system enables high-performance in-memory data access, that is essential to many software applications. Moreover, the project provides a new aspect in database query optimization research by employing programming language techniques.

By the end of the summer, I have finished the implementation of the front-end, the interpreter, and a core of the analysis framework. In the future, I would like to enhance the analysis framework to support more complex data-flow analysis.

Acknowledgements

This project was done as a summer research project. I would like to thank Hanfeng Chen for his guidance in the array programming and in-memory database research fields, Professor Laurie Hendren for her encouragement throughout the summer, and NSERC for financial support.

References

[1] 2017. MonetDB. (2017). https://www.monetdb.org/Home

[2] ANTLR. 2014. ANTLR. (2014). http://www.antlr.org

[3] Jeffry A. Borror. 2008. *Q For Mortals: A Tutorial In Q Programming* (first ed.). CreateSpace.

[4] Samuel Cremer, Michel Bagein, Saïd Mahmoudi, and Pierre Manneback. 2016. Efficiency of GPUs for Relational Database Engine Processing. In *Algorithms and Architectures for Parallel Processing - ICA3PP 2016 Collocated Workshops: SCDT, TAPEMS, BigTrust, UCER, DLMCS, Granada, Spain, December 14-16, 2016, Proceedings (Lecture Notes in Computer Science)*, Jesús Carretero, Javier García Blas, Victor Gergel, Vladimir V. Voevodin, Iosif Meyerov, Juan A. Rico-Gallego, Juan Carlos Díaz Martín, Pedro Alonso, Juan José Durillo, José Daniel García Sánchez, Alexey L. Lastovetsky, Fabrizio Marozzo, Qin Liu, Md. Zakirul Alam Bhuiyan, Karl Fürlinger, Josef Weidendorfer, and José Gracia (Eds.), Vol. 10049. Springer, 226–233. https://doi.org/10.1007/978-3-319-49956-7_18

[5] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45. http://sites.computer.org/debull/A12mar/monetdb.pdf

[6] Shams Mahmood Imam, Vivek Sarkar, David Leibs, and Peter B. Kessler. 2014. Exploiting Implicit Parallelism in Dynamic Array Programming Languages. In *ARRAY'14: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, United Kingdom, June 12-13, 2014*, Laurie J. Hendren, Alex Rubinsteyn, Mary Sheeran, and Jan Vitek (Eds.). ACM, 1–7. https://doi.org/10.1145/2627373.2627374

[7] Roy E. Lowrance. 2009. How Fast Can APL Be? (feb 2009). http://www.cs.nyu.edu/manycores/pres.pdf

[8] MonetDB. 2017. Functions. (2017). https://www.monetdb.org/Documentation/Manuals/MonetDB/MAL/Functions

[9] MonetDB. 2017. Polymorphism. (2017). https://www.monetdb.org/Documentation/Manuals/MonetDB/MAL/Polymorphism

[10] MonetDB. 2017. User Defined Functions. (2017). https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/UserDefinedFunction

[11] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1493–1508. https://doi.org/10.1145/2723372.2747645

[12] Dennis Shasha. 2002. K as a Prototyping Language. (2002). http://www.cs.nyu.edu/courses/fall02/G22.3033-007/kintro.html

[13] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki (Eds.). ACM, 145–156. https://doi.org/10.1145/564691.564709