Do Programming Languages need Query Languages?

Jelle Hellings

Department of Computing and Software
McMaster University
1280 Main Street West, Hamilton, ON, Canada

1 Introduction

Data processing plays a central role in many general-purpose programs. This central role is underlined by the functionality included in standard support libraries provided by modern programming languages. Take, for example, the C++ standard library: this library includes several efficient *data structures* to represent data collections and a plethora of *algorithms* to operate on these data collections. Indeed, the algorithms in the C++ standard library can even be used to perform all elementary relational algebra operations. Furthermore, the recently added `<ranges>` functionality even allows for the high-level expression of data processing operations via *views*.

To illustrate a high-level data processing task using *views*, consider the following program that *queries* for parents of children living in Hamilton:

```
using namespace std::views;

auto where_pred = [](auto l)
    { return l.place == "Hamilton"; };
auto product_pred = [](auto t)
    { auto [po, p] = t;
      return po.child == p.name; };

for (auto [po, p] : cartesian_product(
            parents,
            persons | filter(where_pred)) |
                filter(product_pred)) {
    std::cout << po.parent << std::endl;
}
```

Similar functionality exists in most major programming languages, e.g., LINQ in C# and other .NET languages, Streams in Java, and list comprehensions in Python.

Although these data processing functionalities do provide the ability to express complex queries, they do not guarantee performance: it is up to the programmer to ensure an efficient program. A programmer can do so by selecting the proper ways to structure, store, and maintain the data; the proper operations to perform on the data; and the most efficient order of these operations. In the example provided above, the programmer made several poor decisions, e.g., by excessive copying and by performing a potentially-expensive *Cartesian product*.

The attention to detail required for a performant data processing program in C++ is in sharp contrast to how any database system with a high-level query language operates. For example, in a Datalog-based database system, one could express the above query via the query:

```
Result(parent) :- parents(parent, c),
                  persons(c, "Hamilton").
```

The above Datalog query is significantly simpler than the C++ program provided before. Furthermore, it is highly likely that the database system will produce a query evaluation strategy that is close to optimal (e.g., by using any indices available on parents and persons) and much more efficient than the approach expressed by the C++ program.

2 Problem Statement

Not all data processing tasks happen in an environment in which a database system is available. Hence, shifting data processing tasks toward database systems for efficiency reasons is not always an option.

As an alternative, we propose a support library via which one can *embed* high-level database-like data abstractions and queries within the program (as-if these where any ordinary data structure or algorithm). In specific, our support library embeds support for Datalog queries and for specifying data structures that manage relational data (including primary key and foreign key constraints) into C++.

A crucial part of our approach is the development of a *compile-time query optimizer* than can produce highly-efficient query evaluation algorithms given only the information available when compiling source code: the queries itself and the data structures on which these queries will be evaluated. By performing query optimization *once* at compile-time, we are able to apply the *zero-cost principle* within our proposed library by eliminating any unnecessary overheads and, hence, provide performance close to (or even surpassing) carefully-crafted data processing algorithms.