Code generation for efficient query processing in managed runtimes

Fabian Nagel
University of Edinburgh
F.O.Nagel@sms.ed.ac.uk

Gavin Bierman*
Oracle Labs
Gavin.Bierman@oracle.com

Stratis D. Viglas
University of Edinburgh
sviglas@inf.ed.ac.uk

ABSTRACT

In this paper we examine opportunities arising from the convergence of two trends in data management: in-memory database systems (IMDBs), which have received renewed attention following the availability of affordable, very large main memory systems; and language-integrated query, which transparently integrates database queries with programming languages (thus addressing the famous 'impedance mismatch' problem). Language-integrated query not only gives application developers a more convenient way to query external data sources like IMDBs, but also to use the same querying language to query an application's in-memory collections. The latter offers further transparency to developers as the query language and all data is represented in the data model of the host programming language. However, compared to IMDBs, this additional freedom comes at a higher cost for query evaluation. Our vision is to improve in-memory query processing of application objects by introducing database technologies to managed runtimes.

We focus on querying and we leverage query compilation to improve query processing on application objects. We explore different query compilation strategies and study how they improve the performance of query processing over application data. We take C$^\sharp$ as the host programming language as it supports language-integrated query through the LINQ framework. Our techniques deliver significant performance improvements over the default LINQ implementation. Our work makes important first steps towards a future where data processing applications will commonly run on machines that can store their entire datasets in-memory, and will be written in a single programming language employing language-integrated query and IMDB-inspired runtimes to provide transparent and highly efficient querying.

1. INTRODUCTION

Over the last two decades, DRAM prices have been dropping at an annual average of 33% with this trend projected to continue. As of April 2014, enterprises can buy servers with a DRAM capacity of more than 1TB for under US$50,000. This trend has led to the emergence of in-memory database systems, which keep the entire dataset in main memory to allow for more efficient query processing. Such deployments, useful as they may be, still require the application programmer to think at two different levels: the application level, with data expressed in the data model of a host programming language; and the data manipulation level, with data expressed in the relational model and processed through SQL. There is no synergy between these two runtimes; the programmer needs to either take care of the impedance mismatch through manual and error-prone data model translation, or through a high-level API that bridges the two data models and manipulation methods.

The impedance mismatch has led to the introduction of language-integrated query, which has been revived, in part because of its support in Microsoft's LINQ framework [13]. Language-integrated query presents a single, uniform framework to query data both in the memory space of an application as well as in external data sources such as database systems. Using a query language that is integrated into the programming language offers various benefits to an application developer over the traditional approach of formulating SQL queries as string literals and then submitting the query to a database system using middleware APIs such as ODBC or JDBC. Language-integrated query is transparent to the developer by providing a query syntax that is native to the programming language and by representing externally stored data in the data model of the host programming language. However, for data that is stored in an external DBMS, the developer is limited to the expressive power of the relational model whereas for all other data she can harness the full power of the host programming language (*e.g.,* nested data, inheritance). Furthermore, it is not (efficiently) possible to use imperative constructs to process data stored in an external DBMS or to keep references to elements in the database. Therefore, the developer might prefer to store data in the memory space of the application instead of in an external DBMS. More so if the application does not need the bloat of a DBMS server [10], *e.g.,* because there are no concurrent applications accessing the data and, hence, no need for transactions. Such applications often employ static schema definitions and queries that are constructed from a limited number of predefined query patterns and whose instances only vary in a few parameters (*e.g.,* a selection predicate) based on user interaction (*e.g.,* via GUI elements). These queries are an excellent opportunity to be compiled to native code either through just-in-time compilation or as a library to be embedded in the application.

In this work we explore database-inspired strategies to make query processing on data elements in the managed memory space of a host programming language more efficient by leveraging query compilation [12, 16, 19]. Query compilation is a highly efficient and easily realisable way to improve the performance of language-integrated query in managed runtimes. We propose to store all data

*This work was done at Microsoft Research, Cambridge.

in the memory space of the programming language to give the application developer direct access to the dataset, but to move the heavy lifting of query processing to a different runtime, here native C code, that is better geared towards the task. Our techniques could be applied to process queries in any modern object-oriented language, but for concreteness we will focus here on query processing using LINQ as the querying language and C$^\sharp$ as the programming language. We present three approaches to improve the performance of queries on data in the memory space of a managed application. The baseline approach compiles the query statement to optimized C$^\sharp$ code for processing the query. We then focus on a special case where the use of arrays of C$^\sharp$ `struct`s allows us to directly process the query in native C code. This approach enables row-oriented data storage and, hence, permits processing strategies that are closer to those of database systems. Finally, we propose processing generic C$^\sharp$ queries using a combination of compiled C$^\sharp$ and C code. The compiled C$^\sharp$ code stages the input data to allow the compiled C code do the heavy lifting of the query. We believe our approaches pave the way towards a tighter integration between the two runtimes by carefully introducing main-memory database query processing primitives into host programming languages. Our main contributions, and a roadmap of the rest of this paper, are:

- We outline the main inefficiencies that are inherent to processing queries on in-memory collections using language-integrated query and LINQ in particular (§2).
- We explore ways for improving the efficiency of querying data in the memory space of a managed application by leveraging query compilation and techniques from IMDBs (§3).
- We present three novel approaches to the problem that rely on compiling (parts of) the query into native C code (§§4–6). We look at a broad spectrum of techniques, from generating efficient host programming language code, to offloading the entire computation to an efficient C runtime, to combining the two extremes. We show how these approaches can be implemented and what benefits and drawbacks they have. The approaches are optimized for both response time and cache behavior—though to a different extent depending on the specific approach—as is expected in an in-memory setting.
- We evaluate our approaches on LINQ-to-objects, the default query processor for in-memory objects in C$^\sharp$. We use both response time and cache profiling as metrics and show how our approaches lead to significant performance improvements (§7). Our techniques exhibit the potential of achieving query processing performance over native objects of a programming language that approaches that of a tuned IMDB using state-of-the-art techniques stemming from query compilation.

Finally, we present related work in §8 and conclude and identify future work directions in §9.

2. LINQ

Language-Integrated Query (LINQ) is a framework introduced by Microsoft that adds powerful query-like capabilities to C$^\sharp$ and Visual Basic (in this paper we focus exclusively on C$^\sharp$). This is achieved by defining a design pattern of general-purpose query operators and extending both programming languages with special query syntax that is compiled into these operators. The LINQ framework also provides a number of domain-specific implementations of the query operators that enable use of LINQ over in-memory .NET collections (*e.g.,* arrays, lists, etc.), relational databases and XML documents. The framework is designed to be extensible, so developers can create their own domain-specific implementations.

LINQ bridges the semantic gap between programming languages and query languages. Previously, programming languages accessed query engines via a weak embedding, where queries are expressed as strings and are interpreted at runtime by the query engine. This approach has several disadvantages for developers. First, they have to learn a new query language for each type of data source that they must support (*e.g.,* SQL for relational data; or XQuery for XML). Second, there is no support from the programming language to ensure that the embedded query is well-formed, or well-typed. Lastly, this approach is (infamously) insecure: injection attacks are a direct consequence of the naïve representation of queries as strings.

LINQ, in contrast, offers a consistent model for representing and querying various kinds of data sources based on the principles and syntax of the host programming language. Moreover, the query language is deeply integrated into the host language to further support the programmer when creating the data representation and queries for an application. LINQ supports an SQL-like query syntax:

```
var qry_stmt = from s in source where s.Name == 'London'
               select s.Population;
```

This query syntax is merely (convenient) syntactic sugar, as it is compiled away to a series of method calls, e.g.

```
var qry_stmt = source.Where(s => s.Name == 'London')
               .Select(s => s.Population);
```

These methods on the data source (*e.g.,* `Where`) are known as the standard query operators. Many of these operators take lambda expressions (*e.g.,* `s => s.City == 'London'`) as arguments, and some of these methods directly correspond to relational algebraic operations. The methods are overloaded to allow querying different types of data sources using the same syntax. We next describe the two implementations provided by the LINQ framework.

2.1 LINQ-to-objects

All the in-memory .NET collections, *e.g.,* `List<T>` or `Array`, implement the `IEnumerable` or `IEnumerable<T>` interfaces, and implementations of the standard query operators are provided. This implementation is known as LINQ-to-objects. Each operator is implemented as an iterator method that returns an enumerable to iterate over its result (via an `IEnumerator`). This enumerable serves as input to the following iterator method. Iterator methods can be chained to processes the query in an object-at-a-time fashion closely resembling the tuple-at-a-time [8] paradigm of relational query engines. Each enumerable continuously pulls the next object from its input enumerable(s) until it can produce its next output object. It returns the object and once the caller requests the next object from the enumerable, it resumes processing.

The semantics of iterators is lazy—as it stands the code above simply returns the constructed query statement, but does not actually execute the query. Each object of the query statement's result is only produced when the application consumes it (deferred execution). One way to execute the query statement and produce all objects in its result set is to iterate over them in a `foreach` loop. When `qry_stmt` is executed in a `foreach` loop, the loop requests the first object of the query's result from the enumerable of the `Select` method. The `Select` method in turn requests an object from the enumerable of the `Where` method. The `Where` method iterates over the data source until it finds an object that satisfies the filtering condition and returns that object to the `Select` method which performs the appropriate projection on the object and returns the result to the `foreach` loop. This process repeats until all results are produced.

2.2 LINQ query providers

LINQ provides two additional interfaces which are derived from the enumerable interfaces: `IQueryable` and `IQueryable<T>`. These interfaces also support the standard query operators but the chief

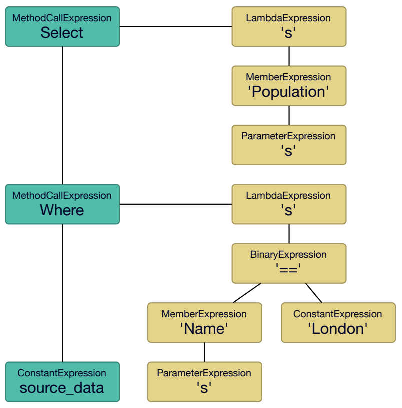


Figure 1: Example of an expression tree

difference is that any lambda expression arguments are quoted, *i.e.,* they are implicitly converted into expression trees.[1] This implicit conversion is implemented by the C♯ compiler.

Implementations of these queryable interfaces are known as LINQ query providers. This is the means by which data-source-specific implementations are defined, typically by interpreting the expression tree to retrieve the query's result from a data source. Expression trees are at the core of the LINQ extensibility model. One example of a provider that is part of LINQ is LINQ-to-SQL which translates the expression tree to a SQL query statement. Executing the resulting query statement actually executes the query on an external DBMS with the results returned as C♯ objects.

The expression tree for the query statement from earlier is shown in Figure 1. It follows the sequence of method calls that would have been executed in LINQ-to-objects. The `ConstantExpression` nodes represent any kind of constant data, which includes the query's input. The `MethodCallExpression` nodes represent method calls (*i.e.,* the standard query operators) and the `LamdaExpressions`, together with their descendants, represent the arguments of the method call.

2.3 Inefficiencies in LINQ-to-objects

Querying data in an application's memory space through LINQ-to-objects exhibits many inefficiencies, more so as the data volume grows. As the source code for the `System.LINQ` namespace classes is closed source, we cannot authoritatively say how LINQ processes queries; however, by looking at the decompiled source code of these classes and running microbenchmarks, we can posit a fairly complete picture of the internals and their inefficiencies. Most of these inefficiencies are inherent to LINQ's execution paradigm and are also found in other implementations (*i.e.,* in Mono).

Execution paradigm LINQ's execution paradigm resembles the tuple-at-a-time paradigm in database systems, and so shares some of its drawbacks, *i.e.,* mainly a high per-element overhead [2, 17]. The enumerable returned by iterator methods is auto-generated by the compiler. When an iterator method iterates over its input enumerable in a `foreach` loop, the compiler emits the following code:

```
IEnumerable<T> enumerable = /* Call Input Method */;
IEnumerator<T> enumerator = enumerable.GetEnumerator();
while(enumerator.MoveNext()) {
  T s = enumerator.Current;
  /* Body of foreach loop */ }
```

Since `MoveNext()` and `Current` are defined in an interface, they are virtual functions. Thus, each iterator imposes the overhead of two virtual calls per input element [14]. In an object-oriented language, virtual calls are expensive, since the compiler cannot inline code at the call site as the receiver method is known only at run-time. Furthermore, iterators contain state machine logic in the `MoveNext()` method, which further adds to the per-element overhead [14].

Generics and lambda expressions LINQ defines a set of standard query operators that are used to compose arbitrary queries. This is achieved by using generic types to specify the output type of iterator methods and by using lambda expressions as parameters of the iterator methods to perform operation and type-specific tasks on each input element (*e.g.,* test a filtering condition). However, replacing all generic types with their respective types and lambda expressions by their equivalent code improves the execution time.

Independent operators LINQ-to-objects does not exploit synergies between successive query operators. Consider a query that contains an `OrderBy` with a subsequent `Take(N)`. LINQ-to-objects first evaluates the `OrderBy` and then returns the first `N` result elements. A better approach would be to merge both operations and maintain a heap with the `N` highest/lowest values instead of sorting the entire input of the `OrderBy` operation.

Aggregation Aggregation is a good example to illustrate missed synergies between query operators. In LINQ, aggregation is expressed by a `GroupBy` method call that groups all input elements by a key and either a result selector in the `GroupBy` or a successive `Select` method call that constructs the result for each key and contains one or more aggregation operations (also method calls) such as `Sum` or `Count`. In both cases, each aggregation iterates over all elements in the group to compute the aggregate. We conducted a simple experiment based on the aggregation in Q1 of the TPC-H benchmark. Our results show that LINQ could process the aggregation 38% faster if it would process all aggregations in a single loop over all elements of the same group. Furthermore, LINQ does not recognize overlaps in the aggregation computations and computes the count of a group for each aggregate computation individually. Eliminating these duplicate computations improves performance by a further 12%. Then, by collapsing the grouping and the aggregate computations in a single loop, it could gain another 10%.

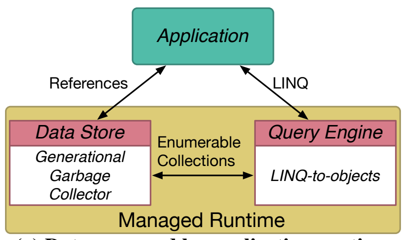
Limited query optimization The query syntax in LINQ resembles SQL but LINQ-to-objects lacks the optimization stages common in relational DBMS due to the lack of semantic information (*e.g.,* schemata, histograms) about the underlying data. The query syntax is usually translated into method calls composed in the order in which operations are declared. This requires the programmer to have an understanding of query processing to write efficient LINQ queries. Query avalanches [4, 9] when dealing with nested sub-queries are one of the unfortunate artifacts of this approach. Despite the lack of semantic information, some heuristic rewrites could still be applied on LINQ queries *e.g.,* selection push-down or reordering selection predicates according to expected processing cost. For instance, forcing the selections of Q3 of the TPC-H benchmark to be applied before the join between `lineitem` and `order` results in a 35% performance improvement; LINQ in no way enforces this more efficient evaluation order.

C♯ LINQ-to-objects is implemented in C♯, whereas database systems are usually written in a lower-level systems language to allow better control over how a query is processed and for performance. To compare, the same quicksort implementation on the same data runs 58% faster in compiled C code over its C♯ counterpart. C♯, furthermore, has additional inherent overheads including the interpretation overhead; dynamic dispatching; and garbage collection.

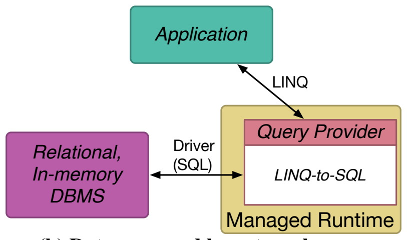
3. OVERVIEW

In a typical managed application, data can either be stored as collections of objects in the memory space of an application (Fig-

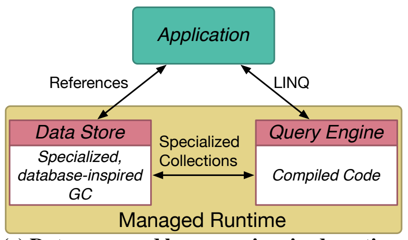
[1]An expression tree is an AST representation of a given query.



(a) Data managed by application runtime



(b) Data managed by external DBMS



(c) Data managed by DBMS-inspired runtime

Figure 2: Strategies for managing queryable datasets

ure 2a) or in a separate database process (Figure 2b). The former allows developers to access data with the expressive power of the programming language (*e.g.,* references, custom loops or LINQ queries) whereas the latter allows them to access data only through a particular query provider (*e.g.,* LINQ-to-SQL). By restricting data access to queries on relations of a predefined schema, databases have more liberty on how to store data, how to optimize queries, and how to deal with concurrent requests. In contrast, querying collections in the memory space of an application is limited by the implementation of the querying language and the storage layout imposed by the runtime's garbage collector. In Figure 2c we illustrate our vision of a database-inspired runtime that allows faster query processing than regular managed runtimes. We do not want to integrate a database system into a managed runtime, but rather to make query processing of collections in the memory space of the application more efficient by leveraging database technology. This allows developers to access and query data elements with the full expressive power of the host programming language at much better performance. We focus on the querying aspects of our vision. We provide new alternatives for querying data inside an application's memory space, *e.g.,* collections of objects or structures, using LINQ. We propose to replace the default LINQ-to-objects execution paradigm that utilizes a pipeline of enumerables with a single, query-specific enumerable that is generated at run-time. We leverage query compilation techniques to generate the source code of the enumerable, then compile and execute it. Query compilation is comparable to query plan generation in a relational DBMS, but generates and compiles highly optimized code instead of an operator plan. Previous work [12, 16, 19, 14] has shown that query compilation results in significantly faster query processing than competing interpretative approaches (*e.g.,* tuple-at-a-time) by providing more predictable and register/cache-friendly memory access patterns; more options for the compiler to perform loop-based optimizations; less interpretation overhead per data element; all these enhancements are present in our techniques. Query compilation is also a good fit with in-memory data processing where keeping the CPU(s) busy is one of the key challenges. Our approach could also be used to *statically* generate the code to answer a LINQ query at compile-time. Doing so would save the cost of generating the binary executable each time the query is run. However, this would require rewriting the C$^\sharp$ compiler, so we leave this to future work.

We use a custom LINQ query provider to dynamically generate, compile, and execute the source code for processing a query. As queries on collections of objects or structures (which implement `IEnumerable<T>`) are automatically processed using LINQ-to-objects, we bypass this behavior by defining wrapper classes that implement `IQueryable<T>` around all collection types that we support (*e.g.,* we wrap the `List<T>` collection with `QList<T>`). The query statement then uses these wrapper classes as source data instead of the underlying collections. Our approach is transparent to developers: application code does not need to be modified more than replacing the C$^\sharp$ collection classes with their functionally-equivalent wrapper collections to use our execution model.

Queries in LINQ are not executed when the application reaches the declaration of a query statement, but when it tries to consume parts of the query's result. However, when the application reaches the declaration of a query statement on a data source that implements the `IQueryable<T>` interface (*i.e.,* our wrapper collections), then its query provider is consulted to return an `IQueryable<T>` that represents the query's result; type `T` refers to the type of the result. This queryable contains a reference to the query's expression tree. We use the expression tree as input for code generation. Figure 3 gives an overview of the processing by our query provider once the evaluation of a query statement is triggered, *e.g.,* by iterating over its result in a `foreach` loop. The `foreach` loop requests an instance of `IEnumerator<T>` from the previously created `IQueryable<T>` object by calling its `GetEnumerator()` method. This object will allow the `foreach` loop to execute the query by iterating over its result. The `GetEnumerator()` call invokes the query provider to obtain an `IEnumerable<T>` object capable of evaluating the query. It then calls `GetEnumerator()` on this enumerable and returns the result to the `foreach` loop. The query provider creates the `IEnumerable<T>` object by generating the code of an iterator method that returns the enumerable, compiling the code, and executing the method.

When generating the source code to evaluate a query, we first traverse the query's expression tree to eliminate all constant subtrees (`ConstantEvaluator`). If the expression tree contains parts that can be evaluated independently of the source data, we evaluate these expressions and replace them with an expression node of the result. This is the canonical representation of the query. After replacing all constant parts, we consult a cache (`QueryCache`) that contains compiled code of previous queries to see if we already have a compiled version of the current query. Queries in the cache are identified by their expression tree. If we have a cache hit the compiled code is executed; otherwise, new code is generated. The system also supports reusing compiled code if the expression trees are essentially the same, but one or more parameters in the query differ (*e.g.,* different values in a selection predicate). Keeping a cache with compiled queries alleviates the non-negligible cost of compiling the generated code [12, 16, 19]. Note that a typical LINQ application does not contain many different query patterns. These are typically hard-coded into the application and the queries only vary in parameter values, which are either generated by the application or supplied by a graphical user interface. Thus, caching compiled code for each query pattern can significantly reduce the compilation overhead and, hence, the query response time.

If the query cache does not contain a compiled version of the query, we must generate and compile the source code to evaluate the query before executing it. We first translate the expression tree into a tree representation of the source code that is to be used to evaluate the query (`ExpressionTreeTranslator`). We then walk this tree to emit source code to be compiled (`CodeTreeTranslator`). The following shows the skeleton of the generated code:

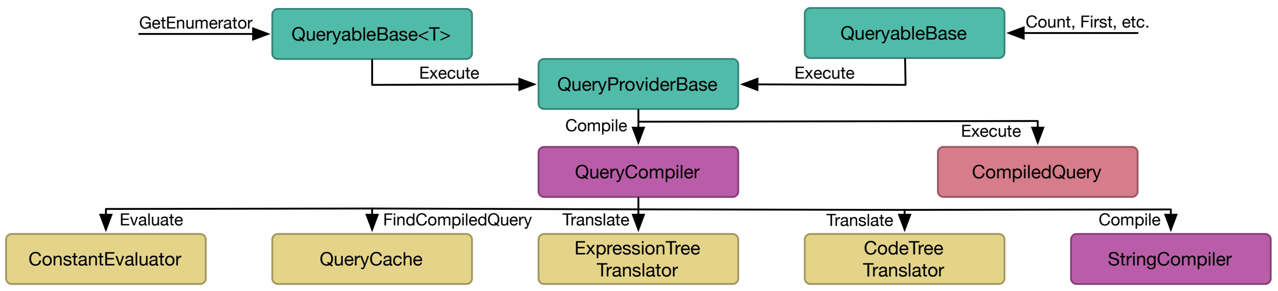


Figure 3: Overview of our query provider

```
// Namespace Declarations
public static class Executor {
  public static IEnumerable<ReturnType> Execute(
           IEnumerable<SourceType> data_1,
           Parameter1 param_1, ...) {
    /* ... */ } }
```

The iterator method `Execute` serves as access point to the compiled code. Once the source code has been generated and compiled, we create a delegate (C♯ function pointer) to allow easy and fast access to the `Execute` method. Executing the delegate returns an enumerable that is capable of evaluating the query by iterating over its result. The query provider then returns the corresponding enumerator object to the `foreach` loop and the `foreach` loop produces the result of the query by iterating over each element in the result.

We have identified three options to leverage code generation for evaluating LINQ queries. The first option is to generate only C♯ code and do all processing in C♯ (§4). Alternatively, we can generate streamlined C code to evaluate the entire query (§5). A third option is to combine the two, where C♯ and C are used synergistically to process queries (§6). We examine each option in turn.

4. GENERATING PURE C♯ CODE

This is a baseline approach that resembles [14]. We generate C♯ code to process the query. However, the generated code does not exhibit the deficiencies of §2.3 (other than the ones inherent to C♯).

4.1 The generated code

To process a LINQ query, we generate C♯ code that is capable of producing the result of the query and encapsulate it in an enumerable to allow the application to iterate over the query's result. The generated C♯ code follows the same principles as previous work on code generation in database systems [12, 16]. We reduce the per-object processing overhead of LINQ-to-objects by replacing the enumerable pipeline with a single enumerable, all generic types with their actual types and all lambda expression calls with their source code equivalent. The code to evaluate a query is structured into one or more tight loops that each incorporate a subset of the LINQ query's operations. This strategy provides more options for the compiler to perform loop-based optimizations and exhibits more predictable and register/cache-friendly memory access patterns. Each loop either produces the final result of a query or an intermediate result of a blocking operation (*e.g.,* aggregation or sort). We only create a single intermediate result per loop construct. For instance, if a loop contains the probing part of several joins followed by an aggregation, then we only create result objects for the combined result and not for each intermediate result, as LINQ-to-objects would. The operations inside each loop are modeled after common database practices (*e.g.,* hash joins). We do not cover parallel execution, but because of our database-centric approach, existing parallelisation strategies [5, 21] are applicable. The following illustrates the generated C♯ code for the example query of §2:

```
public static class Executor {
  public static IEnumerable<int> Execute(
         IEnumerable<SourceType> data_1,
         String param_1) {
    foreach (SourceType elem_1 in data_1) {
      if (elem_1.Name == param_1) {
        yield return elem_1.Population; } }
    yield break; } }
```

All processing is done by one enumerable in a single loop. The equivalent LINQ implementation would require a chain of two iterator methods (`Where` and `Select`) and frequent virtual function calls to transfer data between them. The LINQ implementation would also invoke a lambda expression on each input element to test the filtering predicate and another one on qualifying objects to extract the `Population` member. These lambda expressions are inlined into the generated source code (the `if` and `yield` statements).

4.2 The code generation process

Our query provider implementation uses the expression tree of a query to generate code. After converting the expression tree into a canonical form and checking the query cache, we translate the operations defined by the method call nodes in the expression tree into a tree representation of the source code to evaluate the query. We first traverse the expression tree to identify parts that can be combined into a single loop construct. This is not possible for all operations in the expression tree as blocking operations end the current loop and require to start a successive one. In this case, we use the type specified in the expression tree (result type of the corresponding `MethodCallExpression`) to create objects of the intermediate result that connects both loops. This allows us to access the members of each object as depicted in the expression tree. We do not create any other objects representing intermediate results from the expression tree inside a loop segment, but instead apply pending operations on the objects of the loop's result. To aid the creation of the code tree for each loop segment, we track the names of all variables that we assign to the inputs of the loop (using numerical identifiers) together with the nodes in the expression tree that define each input. These inputs include the current object in each loop iteration and all objects retrieved inside the loop (*e.g.,* from probing a hash table). We further track all assignments that the expression tree defines for operations that are part of the loop segment and the name of the variable storing the objects that embody the result of the loop. This allows us to interleave the processing of the operations defined in the expression tree that we combine into the loop construct.

Each node of the generated code tree represents a fragment of code; the nodes are arranged to reflect fragment order. A node's children are code fragments to be placed in the body of the operation that the node represents (*e.g.,* the body of a `foreach` loop). The order of siblings dictates the code emission order. The information stored about the code fragment at each node is mostly derived from lambda expressions in the expression tree. In Figure 4 we show the code tree that corresponds to the expression tree of Figure 1 and

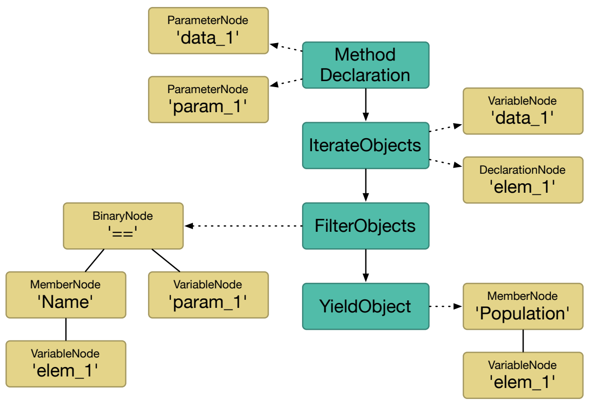


Figure 4: Code tree that is generated from the expression tree of Figure 1.

translated into the source code in §4.1. `IterateObjects` creates the variable for accessing the current element of the loop (`elem_1`) and `FilterObjects` and `YieldObject` access this variable by name.

We then traverse the code tree and generate source code. The C♯ class `CSharpCodeProvider` provides access to instances of the C♯ code generator and compiler. Its `CompileAssemblyFromSource()` method allows us to compile the generated source code in-memory, without having to utilize any external processes. C♯ also supports other means to create executable code, *e.g.,* directly emitting *Common Intermediate Language* (CIL) instructions. We chose to compile from source as it provides us with the full set of tools that any other C♯ implementation has and is more readable than CIL. However, the second step of the code generation phase could be replaced with any method that translates source to executable code regardless of any method-specific intermediate representations.

5. SUBSTITUTING C FOR C♯

In many situations, compiled native C code can outperform general purpose C♯ code running in a managed environment. However, to be able to process LINQ queries in C, we have to be able to access C♯ data from C. While modern managed runtimes provide some means to make object-oriented data accessible to native code, they do not allow access to arbitrary collections of objects. The reasons for this span from safety concerns to more practical issues related to garbage collection. Garbage collectors are free to move objects around in the managed heap and, hence, only objects that are explicitly pinned (not allowed to be moved) and that do not contain references to other, non-pinned objects can be made available to native code. Fortunately, there are cases where LINQ queries can be processed by native code. In C♯, `struct`s are considered value types. Hence, an array of `struct`s stores the data elements at each array position instead of a reference. Storing the source data in fixed-length arrays of `struct`s without references leads to consecutive storage of data in memory and to a flat representation of each data element, comparable to a row-store in a database system. .NET allows to make such arrays accessible to native code. This approach lets us process the data as if it was stored in an in-memory row-oriented DMBS, while still allowing C♯ to directly access it. We further restrict LINQ query definitions to ensure that the query can be evaluated in C. A query statement may not define calls to methods defined in the application (including non-default constructor calls) and may only use supported types (no objects or `struct`s that contain references) as intermediate results. We will now explore the use of native C code to greatly improve the evaluation time of queries that are posed on such, database-inspired, arrays.

5.1 The generated code

Even though query processing is performed by native C code, we still need to generate a C♯ enumerable to allow application code to iterate over the result. The enumerable acts as a wrapper around the C call that processes the query. The following fragment illustrates the generated C♯ code for the query statement example from §2:[2]

```
public static class Executor {
  [DllImport("query0.dll", ...)]
  public static extern int EvaluateQuery(IntPtr ctx);

  public static unsafe
  IEnumerable<int> Execute(IntPtr data_1,
                           IntPtr param_1) {
    Context* ctx = CreateContext(data_1, param_1);
    while (EvaluateQuery(ctx) > 0) {
      yield return ctx->out_elem; }
    yield break; } }
```

The C function called to evaluate the query is `EvaluateQuery`. C♯'s *platform invoke services* (*PInvoke*) allow us to declare it inside the class definition and then call it as if it was a standard C♯ method. The `Execute` method receives a pointer to the source data and the query's parameters. It first allocates and initializes a *context* structure that contains all relevant information for accessing the input data, parameters for evaluating the query, and fields that describe the current state. The `EvaluateQuery` function is called to retrieve the next element of the query's result which can be accessed through the `out_elem` field of the context structure. This evaluation strategy allows support for the deferred execution principle of LINQ: only parts of the query that are consumed by the application have to be evaluated. If the result elements of a query are `struct`s, the `out_elem` field of the context structure will contain a pointer to the next result element. As `struct`s in C♯ are value types, we do not directly return result elements to the caller to prevent C♯ from copying them. Instead, we return a pointer to the result element as `IntPtr` (C♯'s pointer type) and cast it to the correct type in the caller. This significantly reduces the cost of queries with huge results as the result is no longer copied to return it.

The generated C code follows the principles of the generated C♯ code in §4 and exhibits the same optimizations. The following is the C code of the `EvaluateQuery` function to evaluate the query:

```
int EvaluateQuery(Context* ctx) {
  int i;
  for (i = ctx->curr_elem; i < ctx->input_sze; i++) {
    Input* elem_1 = &(ctx->input[i]);
    if (elem_1->City == ctx->param_1) {
      ctx->out_elem = elem_1->Population;
      ctx->curr_elem = i + 1;
      return 1; } }
  return 0; }
```

Similar to the generated C♯ code for the same query in §4.1, we iterate over all elements in the input array, filter them, and return the value of the `Population` field. The value is returned by writing it to the `out_elem` field of the context structure before returning. The `curr_elem` field in the context structure is used to split the iteration over the source data into several function calls. Before returning, `curr_elem` is set to the next element of the input array to continue processing from that element in the next `EvaluateQuery` call.

Memory allocated in C ceases to exist after the C function returns. Thus, we must save state shared between different calls to `EvaluateQuery` (*e.g.,* pointers to hash tables). We do so in the context structure so all invocations of `EvaluateQuery` can access them.

[2]For better readability, we omit some C♯ intrinsics that do not impact the general approach.

5.2 The code generation process

Generating C code for LINQ is similar to generating C♯ code. By traversing the LINQ expression tree, we create a separate tree representation of the C code. The basic approach for generating the code tree remains as outlined in §4.2. However, where we have previously been able to access the source data and (intermediate) results as depicted in the expression tree, the definitions of these structures do not exist in the generated C code. We utilize C♯'s reflection API to recreate these structure definitions. Thus, there is no need to create a separate mapping to translate data accesses defined in the expression tree into equivalent C code. Using (arrays of) `struct`s to access the source data and intermediate results in C improves readability of the generated C code by avoiding unnecessary pointer arithmetic and type casts to access the data. As all intermediate results are only used by the generated C code, we can modify the layout of the fields in their structure definition to improve the query's evaluation time. For instance, we can place fields in a structure that are frequently accessed together (*e.g.,* group-by keys in an aggregation) in close proximity or fields that do not change between two successive intermediate results next to each other to allow to block-copy them between both results. Such optimizations are frequently encountered in IMDBs but are not widely applied in query processing or just-in-time compilation in the context of managed runtimes, thereby bringing another level of performance enhancement to LINQ. While building the code tree, for each loop that produces an intermediate result, we check if there are fields in the result that are either not modified in the loop or are accessed together in the successive loop; in both cases we group them together. Note that the definition of the context structure is also part of the generated code as it has to be adapted to each individual query.

In contrast to C♯ compilation, the compilation of the C code cannot be performed in-memory or without invoking external processes. We compile the C code by placing the generated source code into a file and creating a new process that calls an external compiler to compile the file into a dynamic library. This library is linked into the generated C♯ code by using C♯'s platform invoke services to define a method for calling the generated C function.

6. COMBINING C♯ AND C CODE

Ideally, we would like to lift all type restrictions that have been imposed in §5 and process all LINQ queries with C code. However, .NET forbids direct access to arbitrary collections of objects in the managed heap. Instead, we propose to copy the part of the input that is relevant for processing the query into a representation that can be accessed by native code and then evaluate the query or a part of it using C code. As only data that is relevant for processing the query is copied to the native representation, the copied data exhibits better spatial locality than in the previous approaches, which allows us to leverage cache-conscious query processing techniques.

6.1 The generated code

For simple select-project queries, the cost of copying the source data into a representation that can be accessed from C outweighs the performance benefits of processing in native code. Thus, we only generate C♯ code for such queries; however, for queries that contain complex operations like aggregations, sorting, or joins, we process the most expensive parts in C. A salient decision is whether we fully materialize the data to be exchanged between the two runtimes, or incrementally push processing from C♯ to C using a buffer.

6.1.1 *Full materialization*

In this approach, we copy the data into unmanaged memory so native code can access it. We only copy data that will be processed by C rather than the entire input. The data is copied to a linked list of buffer pages and the generated C code is only invoked once data has been staged. The parts of the query to be processed by native code and, hence, the data to be copied depend on the query. Queries with single operations (*e.g.,* sort) may require less data movement than complex queries. The following is the `Execute` method of the generated C♯ code pattern for the most common case:

```
public static unsafe IEnumerable<ReturnType> Execute(
                  IEnumerable<SourceType> data_1,
                  String param_1) {
  Context* ctx = CreateContext();
  CInput* buffer = AddBuffer(ctx);
  int count = 0;

  foreach (SourceType elem_1 in data_1) {
    if (elem_1.Name == param_1) {
      if (count == ctx->elems_per_buffer) {
        buffer = AddBuffer(ctx);
        count = 0; }
      buffer[count].key = elem_1.key;
      buffer[count].price = elem_1.price;
      count++; } }

  while (EvaluateQuery(ctx) > 0) {
    yield return new ReturnType(ctx->out_elem); } }
```

Here, we process most of the query in native code. To reduce the number of objects copied to unmanaged memory, however, we apply all filtering operations in C♯. The generated code injects an implicit projection that copies only the members of the source objects that will be accessed by native code (`key` and `price`). To copy, we produce one or more linked lists of buffer pages that are allocated in unmanaged memory and, hence, accessible in C. We cast the data part of each buffer page to an array of primitive C♯ type (*e.g.,* `int` or `Decimal`); or an array of a custom structure type that is defined in the generated code. The former represents columnar, the latter row-wise storage. Once all data is staged, the C code is called as was shown in §5. The generated C code follows similar principles as in §5. For every result element returned from C, we create a result object. We assign its members from the native representation of the result elements and yield the object to the caller.

The code pattern above assumes that result objects can be constructed from the output produced by native code; this is not always possible. Consider the case of the query results containing references to objects of the input; or when we have only partially copied data into unmanaged memory (*e.g.,* through implicit projections). In such cases we use the original objects to construct the result rather than copies of them. This ensures that our results are consistent with the results of LINQ-to-objects which allows an application to modify elements of the source collection that have been retrieved using a LINQ query. To achieve this we create a C♯ array of all source objects that satisfy all filters in the query before copying data to unmanaged memory. We then also copy the object's index in the array to unmanaged memory so we can later retrieve the corresponding source object from the array and use it to construct the result object. Consider sorting all elements in a collection. When LINQ-*to-objects* processes the query, it first creates an array that contains references to all objects, an `int` array that contains the indexes of all objects, and an array that contains the keys to sort by. The latter two arrays are passed to a quicksort algorithm to sort the indexes. Our approach would do the same, but then give C access to the index and key arrays and execute quicksort in native code.

6.1.2 *Buffered materialization*

The previous approach results in a large memory footprint. Consider aggregating a huge dataset to reduce it to a couple of aggregate objects. The approaches of §4 or §5 do not require much memory as the hash table for the aggregation only contains the resulting couple of objects. The approach of §6.1.1, however, must first copy all relevant data to unmanaged memory thus increasing the memory footprint. To keep the memory footprint fixed we call the generated C code to process the content of a buffer page once it is full. After all elements in the buffer are consumed by native code, C♯ can overwrite the content of the buffer with the next set of input elements. However, transferring data in a single buffer does not always help to reduce the memory footprint. If the generated C code has to keep all streamed data without any modifications (*e.g.,* streaming the blocking part of a join operation), then we would rather copy everything to unmanaged memory before processing it in C.

The following sample shows the `Execute` method of the generated C♯ code to process a query similar to that of §6.1.1:

```
public static unsafe IEnumerable<ReturnType> Execute(
                   IEnumerable<SourceType> data_1,
                   String param_1) {
  Context* ctx = CreateContext();
  CInput* buffer = AddBuffer(ctx);
  int count = 0;

  foreach (SourceType elem_1 in data_1) {
    if (elem_1.Name == param_1) {
      if (count == ctx->elems_per_buffer) {
        EvaluateQuery(ctx);
        count = 0; }
      buffer[count].key = elem_1.key;
      buffer[count].price = elem_1.price;
      count++; } }
  ctx->streaming_done = 1;

  while (EvaluateQuery(ctx) > 0) {
    yield return new ReturnType(ctx->out_elem); } }
```

Instead of adding a new buffer page to the linked list whenever the current one is full, C is called to process its content. The sample assumes that the generated C code contains a blocking operation and, thus, does not return a result before all input is consumed. We use the context structure to maintain state between different calls to `EvaluateQuery`. The `streaming_done` field of the context structure indicates whether the entire input is consumed. Once this is the case, the generated C code can start producing result elements.

6.2 The code generation process

The source code is generated similarly to §5. As before, we use some of the types defined in the expression tree (result types of `MethodCallExpressions`) to create the structure definitions of the input and output of each loop construct in the C code and of the types C♯ uses to copy the data to and from unmanaged memory. However, the expression tree may now refer to arbitrary C♯ types. Object-oriented languages like C♯ allow for a nested data representation through references to arbitrary data types—even other collections. The generated C code, in contrast, relies on a flat, value-type data representation without references allowing us to leverage relational query processing techniques. The mismatch in representations is a key challenge in processing data in unmanaged memory. We address this issue by creating mappings between the object-oriented data layouts of the expression tree and the flat data layouts that the generated C code expects. This allows us to directly translate data accesses depicted in the expression tree to native code that accesses the corresponding data elements in the unmanaged heap.

A mapping consists of two parts: (*a*) an object-oriented representation of the data as found in the expression tree; and (*b*) a native representation of the data layout that we have chosen to use for processing the query in unmanaged memory. The object-oriented

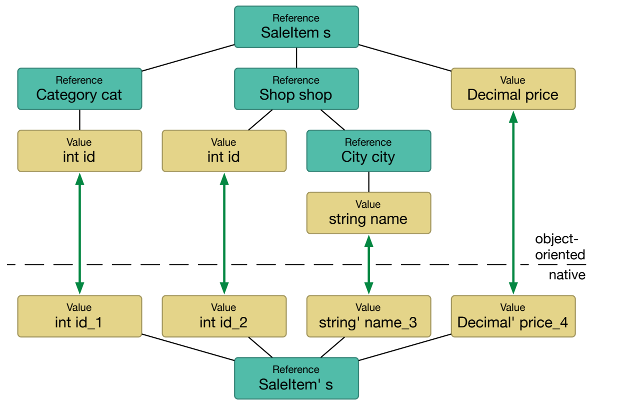


Figure 5: Mapping between C♯ and C value types

data layout is represented by a tree with nodes of four types: *value*, *reference*, *enumerable value* or *enumerable reference*. Value types represent non-composed types such as `integer`, `float` or `string` values. Reference types represent composed types such as `class`es or `struct`s. The children of reference types represent their (public) members. Both enumerable types represent enumerable versions of their respective types. The native data layout is represented by one or more trees that each can either be a single value type node or a reference type node that only contains value type children.

The native representation is usually similar to its object-oriented equivalent, but with all references flattened out, leading to a row-wise data layout in unmanaged memory. Value type nodes of the object-oriented representation map to value type nodes of the native representation (*e.g.,* in Figure 5). However, there are cases where the representations diverge. For example, if some elements in the object-oriented representation are not copied to unmanaged memory but instead represented by an index to a C♯ array that allows C♯ to look them up (see §6.1.1). Then, the reference type to represent these elements maps to the value type that represents the index.

The mappings are created in the same bottom-up pass over the expression tree that constructs the tree representations of the source code. Before generating the tree representation of the code that corresponds to a `MethodCallExpression` node, we have to obtain the mapping for its result. We decide, based on the type of the `MethodCallExpression`, whether to use the same mapping as its child (*e.g.,* `Where`, `OrderBy`) or create a new one (*e.g.,* `Select`, `Join`). In the latter case, one of the `LambdaExpressions` usually specifies the creation of the method's result either by defining a constructor call to produce result objects or by providing a projection that extracts them from the result of its child. We use these definitions to create the object representation of the method's result. Based on the type of the method call and its parameters, we decide how to process it and create the corresponding native representation of its result. We use the native representation of an intermediate result to create its structure definition. For readability, we name the fields of the structure as their equivalent in the object-oriented representation, but append a unique identifier to the name to avoid collisions.

As mentioned in §6.1.1, we inject an implicit projection step into the generated C♯ code before staging the input data to reduce the data volume that has to be copied to unmanaged memory. This projection is driven by the mapping of the type that is copied *i.e.,* the type of the input. The input may contain members that are neither accessed in the query nor part of the query's result. We only add members to the input mapping that are required for processing the query in native code. When coming across a `ConstantExpression` that represents an input collection in the bottom-up traversal of

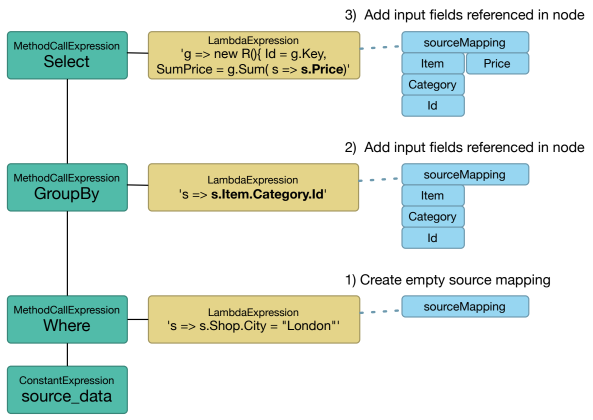

3) Add input fields referenced in node
MethodCallExpression Select
LambdaExpression 'g => new R(){ Id = g.Key, SumPrice = g.Sum(s => s.Price)'
sourceMapping
Item
Price
Category
Id
2) Add input fields referenced in node
MethodCallExpression GroupBy
LambdaExpression 's => s.Item.Category.Id'
sourceMapping
Item
Category
Id
1) Create empty source mapping
MethodCallExpression Where
LambdaExpression 's => s.Shop.City = "London"'
sourceMapping
ConstantExpression source_data

Figure 6: Steps of creating a source mapping (bottom-up; selection ignored because it is performed in C♯)

the expression tree, we create an empty reference mapping for it. Whenever a `LambdaExpression` in one of the expression's ancestor `MethodCallExpressions` references an element of the source data, we check if that element has already been added to the source mapping; if not, we add it. Figure 6 illustrates this process. We perform the implicit projection step by only copying parts of the input to unmanaged memory that are contained in the mapping.

Finally, we use reflection to check if the mapping for the query result has any references to source objects whose members have not been accessed in the expression tree and, hence, have not been added to the mapping. In this case, we extend it with an index to allow us to look them up in C♯ after processing the query in C.

7. EVALUATION

We evaluate the approaches that have been presented in the previous sections for several LINQ queries. All queries are run over a 1GB TPC-H dataset (*i.e.,* a scale factor of one) loaded into the memory space of the application. This use-case models how a programmer would work with a single data representation accessible by programming language constructs (*e.g.,* references); and also be able to formulate complex data processing queries through LINQ.

The first step towards running queries on TPC-H inside an application's memory space is to choose data structures to represent the dataset in C♯. We tested fixed-length `Array`s and `List`s (C♯'s implementation of dynamic arrays) to represent each relation; and objects or `struct`s to represent a record in a relation. Using `struct`s is a poor choice as they are value types in C♯. Thus, when processing a query in LINQ, each struct is copied by value rather than by reference, which significantly slows down query processing. The choice between lists and arrays of objects was much closer with arrays performing slightly better. In the end we chose `List`s for their ability to allow variable sizes. The SQL types of each column were transformed into equivalent C♯ types. For the generated C code we use an array of structs as discussed in §5 to give the generated C code direct access to the data without any data staging.

All tests were performed on a dual core Intel i5-2415M processor with 8GB of memory. Before each test we called the garbage collector and waited until it was finished to ensure that we started with a minimal memory footprint. We first micro-benchmark aggregation, sorting, and join processing individually and then study the performance of queries that combine several of these operations.

7.1 Aggregation

We evaluate the aggregation performance on the aggregation of TPC-H query Q1. We chose this query as it is aggregation-heavy

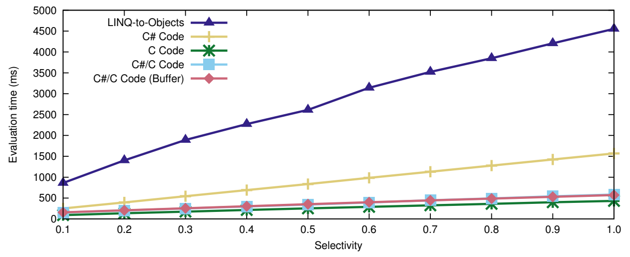


Figure 7: Aggregation over selection; varying selectivity

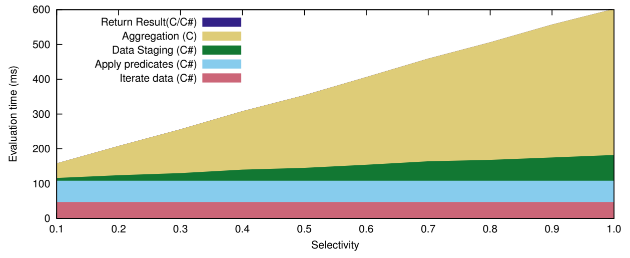


Figure 8: Aggregation cost break down for compiled C♯/C code

and exhibits all the shortcomings of the LINQ-to-objects paradigm (see §2.3). We keep the selection predicate, but vary the selectivity to show how each approach copes with a growing data volume to be aggregated. In Figure 7 we show that all our approaches perform significantly better than LINQ-to-objects; in the case of generated C code even up to one order of magnitude better. As the volume of data to be aggregated grows, LINQ-to-objects looses ground even further. Most of the difference between LINQ-to-objects and the compiled C♯ code is due to the inefficiencies of LINQ-to-objects. The most severe inefficiency is that LINQ-to-objects computes every aggregate in a separate loop over all elements of each group. The generated C code performs the best, which is no surprise: all data is stored row-wise in consecutive memory addresses and all query processing is performed in C without any data staging. This approach resembles the code that would have been generated for a row-wise relational database without indexes, clustering, or a histogram. Combined C♯ and C code performs around 30% to 70% worse than pure C code. This is due to the combined approach having to stage the data in C♯ before shifting processing to C.

We next studied the two different approaches for combining C♯ and C code generation: full or buffered intermediate data materialization. Both perform similarly, but buffering performs slightly better. When aggregating all six million tuples in the dataset, the former approach consumes 390MB of memory for staging the data, whereas the latter only requires the constant buffer size. We tested different buffer sizes to see how sensitive our approach is to this choice, but did not find any significant impact on performance; we therefore settled for a modest buffer size of 64KB. We tested both approaches with performing the selection either in C♯ before staging the data; or in C after staging the data. Applying the selection predicate in C♯ resulted in better performance, in particular for lower selectivities. We further varied the number of aggregates (*e.g.,* `Sum`) that have to be computed while leaving the amount of data that has to be staged constant. The compiled C♯ and C code already outperform LINQ-to-objects if the aggregation only contains a single `Sum` operation. As the number of aggregates grows, the performance difference between the two approaches grows rapidly.

Finally, we broke down the cost of the generated C♯ and C code and report it in Figure 8 for the variant that first stages the data in C♯ before invoking the C code. The cost of iterating over the input and performing the selections is independent of selectivity. Whereas

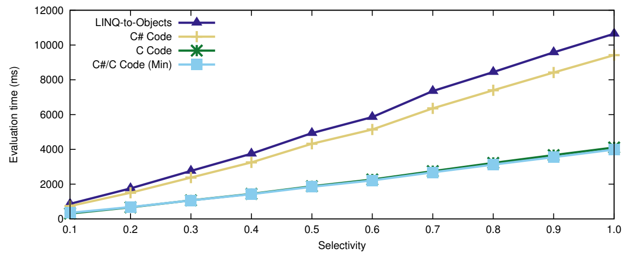


Figure 9: Sorting over selection; varying selectivity

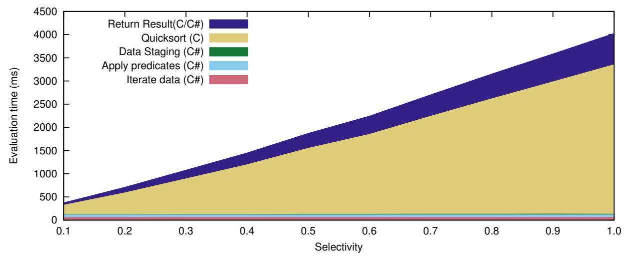


Figure 10: Cost break down of sorting for compiled C$^\sharp$/C code

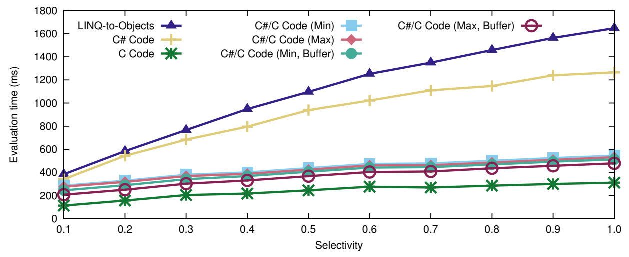


Figure 11: Join over selections; varying selectivity

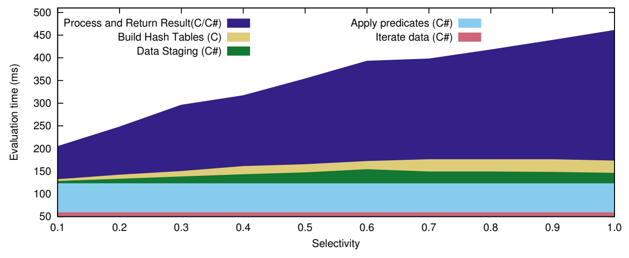


Figure 12: Cost break down of join processing for compiled C$^\sharp$/C code (Max)

the data staging cost grows with selectivity, it does not grow as fast as the aggregation cost. These results suggest that as data volume grows the relative cost of data staging might diminish further.

7.2 Sorting

To measure the performance of sorting, we sorted the `lineitem` relation on `extendedprice`. The performance of all approaches for varying selectivity is shown in Figure 9. As before, LINQ-to-objects performs the worst, though it tracks the performance of C$^\sharp$ code much closer this time. The latter uses the same quicksort algorithm as LINQ-to-objects, so the majority of the processing is equivalent; better performance is due to the inherent cost of LINQ-to-objects. To be fair we implemented the same quicksort algorithm in the generated C code. Thus, the generated C code and the combined C$^\sharp$ and C code perform similarly. In the latter approach, we fully materialize the C input as buffering is inapplicable: quicksort requires full arrays. Though the C-only approach does not require any data staging, its impact on overall performance is limited. To support deferred execution, both approaches must call the C function for every result object. For high selectivities this can be up to six million calls to native code. Despite calls to native code being cheap in C$^\sharp$ they still are several times the cost of a regular C$^\sharp$ call.

When sorting by generating both C$^\sharp$ and C code we must return references to input objects instead of their copies. Thus, we transfer to C an array containing all sort keys and an array containing their indexes; and then sort the index array in C. After C has produced the array of sorted indexes, the generated C$^\sharp$ code looks up the object represented by each index and returns it. A break down of the costs of sorting is shown in Figure 10. The cost of quicksort dominates. As we only transfer the sort keys and their indexes to C, the cost of data staging is smaller than that of aggregation. This is offset by the costs of repeatedly calling C and composing the result in C$^\sharp$ *i.e.,* looking up the elements corresponding to the indexes.

7.3 Joins

We evaluate join performance using the joins and a variant of the selections of TPC-H query Q3. We changed the selections on the `lineitem` and `orders` relations to vary the selectivity, but kept a constant selectivity on `mktsegment` for the `costumers` relation. The result of the query is based on the intermediate result of the join sub-query of Q3. In an object-oriented language, the primary-foreign-key relationship between relations can also be modelled through references and pointer joins. We use a value join here, however, since we want to test the performance of our code generation approaches. As shown in Figure 11, the generated C$^\sharp$ code performs better than LINQ-to-objects, even though it processes the query in a similar way, by avoiding the inherent LINQ-to-objects problems. The generated C code again performs the best overall.

There are four approaches to combining generated C$^\sharp$ and C code. One option (referred to as *Min*) is to transfer the minimum amount of data to C by (*a*) transferring the indexes required for joining the relations, and (*b*) looking up the objects in the C$^\sharp$ code to create the result objects. Another approach (referred to as *Max*) is to transfer to C all data that is necessary to process the query and construct the result. Doing so increases the data staging cost, but does not require the exacerbated look up cost for creating results. Both of these approaches have a buffering variant. Buffering, however, is only beneficial for the `lineitem` data; we build hash tables for both other relations and the hash tables require full materialization. All four approaches perform very similarly, with buffering performing slightly better and full-staging marginally outperforming key/index joins. This suggests that in this case the cost of staging all data is cheaper than the cost of the random index lookups.

We show the cost break down when combining C$^\sharp$ and C code for the (Max, non-buffering) variant in Figure 10. In contrast to aggregation and sorting, the join query does not block for the `lineitem` relation. The C$^\sharp$ code continuously requests the next result. The C code supplies it by iterating over the unprocessed part of `lineitem` and probing the hash tables for qualifying elements. As shown in Figure 10 this cost accounts for the majority of the evaluation time.

7.4 Mixed queries

Finally, we evaluate all approaches on more complex queries with several operations. In Figure 13 we show the evaluation times for the first three TPC-H queries. The following observations are consistent across all queries. The generated C code performs best, followed by the combination of generated C$^\sharp$ and C code. The generated C$^\sharp$ code comes third before LINQ-to-objects. As the queries contain more operations, LINQ-to-objects iterates over a longer pipeline of operations and, hence, transfers more objects

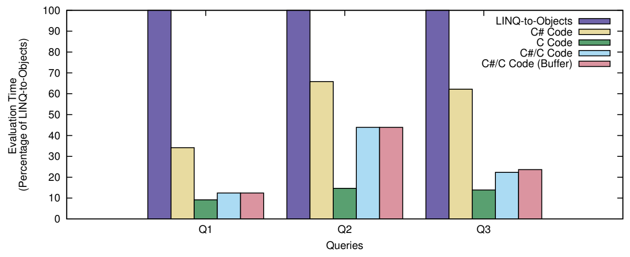


Figure 13: Improvements for TPC-H queries Q1, Q2 and Q3

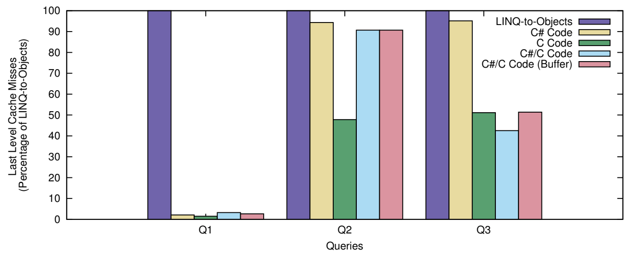


Figure 14: Last level (L3) cache misses for TPC-H queries Q1, Q2 and Q3

through the pipeline and materializes more intermediate result objects, which gives our approaches an additional advantage. The combination of generated C$^\sharp$ and C code has the advantage that the cost of having to stage data before shifting processing to C is often amortized over more operations. For example, sorting after an aggregation does not need any extra data staging. The *Min* approach from earlier is not possible for any of the complex queries tested: C processes more than one operation and requires access to all the data required to construct the resulting objects. Query Q2 contains a nested sub-query. For LINQ-to-objects, we used a hand-optimized query plan that eliminates the nested sub-query to prevent LINQ-to-objects from re-evaluating it for every element and, hence, from significantly increasing the evaluation time [4, 9].

In Figure 14 we show the number of last level cache misses of each approach. All variants of compiled code exhibit fewer cache misses than LINQ-to-objects. Query Q1 shows the greatest benefit as the generated code avoids all additional passes over the input objects that take part in the aggregation (see §2.3); whereas the compiled C$^\sharp$ code for Q3 only shows very little benefit as both execution strategies exhibit similar memory access patterns. The generated C code exhibits the smallest number of last level cache misses for Q1 and Q2 because of the more compact memory layout of its input data. The number of cache misses in Q3 is dominated by hash table probing. As both mixed C$^\sharp$ and C approaches perform an implicit projection step when staging the data, their hash tables are significantly smaller (*e.g.,* the hash table of the `customer` relation only contains an integer value per key) and are more likely to be partially cache-resident when probing the hash tables, hence, reducing the number of cache misses. In this case, the full materialization variant performs better than the buffering one because it reduces cache pressure by only iterating over the staged `lineitem` input when probing the hash tables instead of interleaving this process with fetching qualifying objects from the input and staging the data. These results suggest that it is possible to apply database-centric optimizations to LINQ queries if the managed runtime is treated as an in-memory database system. At the same time, it reinforces the observation that the line between code generation options is not as clear and hybrid approaches that generate both C$^\sharp$ and C code can outperform native solutions under cache-profile-based metrics.

	Q1	Q2	Q3
SQL Server 2014	10360ms	125ms	2766ms
SQL Server 2014—native	2875ms	—	797ms
VectorWise 3.0	946ms	149ms	176ms
LINQ-to-objects	4570ms	41ms	931ms
Compiled C$^\sharp$/C code	567ms	21ms	208ms

Table 1: Performance comparison to an in-memory DBMS

We have so far not reported the cost of code generation and compilation. This cost can be neglected assuming either: (*a*) static compilation at application compile-time for fixed queries; or (*b*) caching and reusing the compiled code for applications with a few query patterns. However, we report these costs for completeness. Source code generation takes between 30ms and 60ms; C$^\sharp$ code compilation needs around 75ms; and C code compilation takes around 720ms. The latter can be reduced by generating LLVM code [16]. Improving the synergy between runtime and compiler, however, is not our focus. Rather, we showcase the potential of code generation for efficient query processing in managed runtimes.

7.5 Comparison to a database system

A comparison between querying collections in a managed runtime and querying data in a DBMS is not strictly fair as the two environments serve distinct use cases. It is not our intention to claim that our approach outperforms a DBMS. Instead, we show that the improvements obtained by our code generation approach are comparable to those obtained in a DBMS and result in performance similar to a state-of-the-art in-memory DBMS. We report the results for the TPC-H queries as executed on SQL Server 2014 in-memory OLTP (Hekaton [6]) and VectorWise 3.0. The former system generates native C code for SQL queries expressed as stored procedures, thus leveraging techniques similar to ours, but in the DBMS realm. The cumulative results are shown in Table 1. To level the playing field, we did not define indexes or utilize multi-core support. Note that Q2 could not be converted into native code as nested queries and `LIKE` predicates are not supported for the stored procedure functionality of this version of SQL Server. Query compilation in SQL Server results in a three-fold improvement over the interpreted version. Our approach exhibits comparable improvements over LINQ-to-objects with query Q1 showcasing a greater benefit due to inefficient aggregate computations in LINQ-to-objects. Further, our approach is comparable and at times surpasses the performance of a heavily optimized in-memory DBMS (VectorWise). By no means do these results imply that managed runtimes are a substitute for relational DBMSs because they address different use-cases. But seeing as native code over managed objects can potentially outperform an in-memory DBMS by bypassing unnecessary functionality, it makes sense to consider a tighter integration of the two; and query compilation is the enabler to improve performance.

8. RELATED WORK

Query compilation in database systems has been around since the times of System-R [3]. However, in recent years it gained traction in the research community [1, 5, 11, 12, 16, 18, 19, 20, 22] as well as in industrial applications such as ParAccel, Netezza [7] or Hekaton [6]. Rao *et al.* [19] present a relational, in-memory, Java-based database prototype that generates Java code, compiles it into Java Bytecode and loads and executes it through the JVM. Data is stored as Java objects containing primitive types. Krikellas *et al.* [12] propose generating C code instead. The authors precede query processing with a staging phase that prepares the input data for cache conscious query processing. Neumann [16] proposes generating LLVM code to reduce the compilation cost. Additionally, he generates code that maximizes the processing performed in each loop and therefore keeps data in CPU registers as long as possible. Sompolski *et al.* [20] studied the benefit of just-in-time compilation at the query operator level in a vectorized query engine. The DBToaster project [1] uses compiled C++ or Scala code to incrementally maintain internal representations of materialized views. Pirk *et al.* [22] combine the partially decomposed storage model with query compilation to eliminate CPU-inefficient function calls; and Dees *et al.* [5] explore code generation for many-cores in main memory column-stores. We employ state-of-the art techniques from previous work in query compilation and adapt them to the challenges (*e.g.,* restricted data access, slow environment for query processing) of querying objects in a managed runtime.

Murray *et al.* [14] were the first to propose using compiled code to process LINQ queries on in-memory objects. They provided a comprehensive description of their code generation algorithm, which generates code that is similar to our baseline approach (§4). In contrast to our approach, they do not interleave the processing done by LINQ operations in the same loop segment to save materialization costs and use nested loops to process join operations instead of hash joins. We extend their work by providing further approaches (§5 and §6) for significantly faster query processing.

Grust *et al.* [9] performed a comprehensive analysis of the performance of LINQ queries on mixed external data sources (*e.g.,* DBMSs and XML). They identified query avalanches as a key inefficiency of LINQ when dealing with nested queries. They presented a new query provider to optimize LINQ queries and avoid query avalanches. In this work, we do not tackle the inefficiencies of LINQ as a whole, but those that arise from the LINQ-to-objects implementation. Cheney *et al.* [4] give a theory and implementation for language-integrated query that is more robust and efficient than LINQ and prevents query avalanches. This work is complementary and can be used together with our work for better performance.

9. CONCLUSION AND FUTURE WORK

We explored strategies to improve the efficiency of processing queries in the memory space of an object-oriented language by leveraging code generation. We used Microsoft's LINQ-to-objects as a reference point and presented three approaches that—at runtime—use a LINQ query provider to generate and compile query-specific source code to process a query. We discussed how compiled C$^\sharp$ code can improve the evaluation time. We then focused on a special case where fixed-length arrays of `structs` can be used to create a row-oriented data store, which enables processing the query using C code. Finally, we proposed to combine compiled C$^\sharp$ and C code. We used the compiled C$^\sharp$ code to stage data to make it accessible to C code and then generated C code to perform the heavy-lifting of the query. We evaluated our approaches against LINQ-to-objects using queries based on the TPC-H benchmark and showed that they deliver significant performance improvements.

From an ease-of-development viewpoint, the C$^\sharp$-only and the combined approach are far superior as they are the most transparent to the developer, merely requiring to change the collection type used for managing data. The combined approach improves the performance of the C$^\sharp$-only version if the majority of the query processing can be offloaded to the C runtime. However, from a performance viewpoint, the C-only approach delivers the best results. Based on this finding, we conclude that improving the query engine is only a first step and that other parts of the runtime, such as the garbage collector, must also be adapted for database-inspired query processing (*e.g.,* consecutive memory locations for objects in a collection). Further optimizations include the integration of code generation into the just-in-time compiler of the runtime and the introduction of structures such as indexes as well as support for clustering, histograms, parallel execution or query result caching [15].

Acknowledgments The authors would like to thank the anonymous reviewers for their comments. This work was supported by a Microsoft Research PhD Scholarship and the Software for Persistent Memories program of the Intel University Research Office.

10. REFERENCES

[1] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10), 2012.

[2] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *VLDB*, 2005.

[3] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, et al. Support for repetitive transactions and ad hoc queries in System R. *ACM TODS*, 6(1), 1981.

[4] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.

[5] J. Dees and P. Sanders. Efficient many-core query execution in main memory column-stores. In *ICDE*, 2013.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, 2013.

[7] P. Francisco. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Redbook*, 2011.

[8] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE*, 6(1), 1994.

[9] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1-2), 2010.

[10] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[11] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[12] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[13] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[14] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, 2011.

[15] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *ICDE*, 2013.

[16] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.

[17] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

[18] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. CPU and cache efficient management of memory-resident databases. In *ICDE*, 2013.

[19] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using JVM. In *ICDE*, 2006.

[20] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, 2011.

[21] S. D. Viglas. A comparative study of implementation techniques for query processing in multicore systems. *IEEE TKDE*, 26(1), 2014.

[22] Y. Zhang and J. Yang. Optimizing I/O for big array analytics. *PVLDB*, 5(8), 2012.