View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Archive

THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Efficient query processing in managed runtimes

Fabian Nagel

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2015

Abstract

This thesis presents strategies to improve the query evaluation performance over huge volumes of relational-like data that is stored in the memory space of managed applications. Storing and processing application data in the memory space of managed applications is motivated by the convergence of two recent trends in data management. First, dropping DRAM prices have led to memory capacities that allow the entire working set of an application to fit into main memory and to the emergence of in-memory database systems (IMDBs). Second, language-integrated query transparently integrates query processing syntax into programming languages and, therefore, allows complex queries to be composed in the application. IMDBs typically serve as data stores to applications written in an object-oriented language running on a managed runtime. In this thesis, we propose a deeper integration of the two by storing all application data in the memory space of the application and using language-integrated query, combined with query compilation techniques, to provide fast query processing.

As a starting point, we look into storing data as runtime-managed objects in collection types provided by the programming language. Queries are formulated using language-integrated query and dynamically compiled to specialized functions that produce the result of the query in a more efficient way by leveraging query compilation techniques similar to those used in modern database systems. We show that the generated query functions significantly improve query processing performance compared to the default execution model for language-integrated query. However, we also identify additional inefficiencies that can only be addressed by processing queries using low-level techniques which cannot be applied to runtime-managed objects. To address this, we introduce a staging phase in the generated code that makes query-relevant managed data accessible to low-level query code. Our experiments in .NET show an improvement in query evaluation performance of up to an order of magnitude over the default language-integrated query implementation.

Motivated by additional inefficiencies caused by automatic garbage collection, we introduce a new collection type, the *black-box collection*. Black-box collections integrate the in-memory storage layer of a relational database system to store data and hide the internal storage layout from the application by employing existing object-relational mapping techniques (hence, the name *black-box*). Our experiments show that black-box collections provide better query performance than runtime-managed collections by allowing the generated query code to directly access the underlying relational in-memory data store using low-level techniques. Black-box collections also outperform

a modern commercial database system. By removing huge volumes of collection data from the managed heap, black-box collections further improve the overall performance and response time of the application and improve the application's scalability when facing huge volumes of collection data.

To enable a deeper integration of the data store with the application, we introduce *self-managed collections*. Self-managed collections are a new type of collection for managed applications that, in contrast to black-box collections, store objects. As the data elements stored in the collection are objects, they are directly accessible from the application using references which allows for better integration of the data store with the application. Self-managed collections manually manage the memory of objects stored within them in a private heap that is excluded from garbage collection. We introduce a special collection syntax and a novel type-safe manual memory management system for this purpose. As was the case for black-box collections, self-managed collections improve query performance by utilizing a database-inspired data layout and allowing the use of low-level techniques. By also supporting references between collection objects, they outperform black-box collections.

Acknowledgements

I'd like to take the opportunity to thank the many people who supported and guided me on my path towards and during my Ph.D. in Edinburgh. I cannot imaging having gone through this journey without them.

Professor Stratis Viglas was my primary supervisor. He did an excellent job at guiding me through my research and keeping me calm and focused at times when I got frustrated with said research. I learned a lot from him during my studies and he always had an open door when I was looking for valuable advice or just to have a chat. During my studies, he always gave me enough space to find my own path, but, at the same time, I knew that he would gently guide me back on track when I got lost. Beyond my studies, he greatly helped me to find a job. I am very thankful for his guidance and friendship. Dr. Gavin Biermann was my second supervisor. I was very lucky having him as my Microsoft Research supervisor as he was very committed to our work and took the time to have weekly meetings with us to discuss my progress. He gave me valuable technical advice and helped me to find an Internship at Microsoft Research in Cambridge. I am very grateful that he voluntarily stayed on as my second supervisor, even after he left Microsoft Research. I want to thank the University of Edinburgh for giving me the opportunity to do a Ph.D. in Edinburgh. I also want to thank Mary and Armeane Choksi and Microsoft Research for financially supporting my studies.

During my studies, I went on two internships. I had a great time during both internships and learned a lot about research outside of academia. Both internships were a very welcome break from doing a Ph.D. and I returned from them with a fresh mind and new energy for carrying on with my studies. My first internship was at IBM's T.J. Watson Research Center. I am very thankful to my supervisor Dr. Mohammad Sadoghi, the database group and the many great people I met there for making this internship so great. My second internship was with Dr. Aleksandar Dragojevic at Microsoft Research in Cambridge. After we got on well during the internship, he replaced Gavin as my official Microsoft Research supervisor. I am very thankful that he also got very involved in the project and I ended up having two secondary supervisors to guide me through the last year of my Ph.D.

There are also people that led me on the path towards doing a Ph.D. and prepared me for it. Professor Torsten Grust was my database professor at Tuebingen University. His lectures sparked my interest in database engineering and he arranged for me to do my Diploma thesis at VectorWise / CWI in Amsterdam. Professor Peter Boncz supervised my thesis and I quickly found out how lucky I was to have had him as a

supervisor. During my time with Peter, I learned a lot about software engineering, database design and how to conduct research. I think that without Peter's influence, I would never have started a Ph.D. and I would not have had the ambition to look for a job as good as the one I will start after finishing my studies. He introduced me to Stratis and helped me on many other occasions to further my career. I value his friendship and am very grateful for everything he did for me.

During my time in Edinburgh, I spent time with some exceptional people. With Edinburgh being often referred to as the "Athens of the North", it was no surprise that most of them are Greek. I am thankful for the many very funny lunch and tea breaks with my fellow database Ph.D. students Andreas Chatzistergiou and Michail Basios that greatly helped to get distracted from work and provided new energy to get back to it afterwards. Both are great friends and I enjoyed the time I spent with them. We had many joint cooking and dinner events with them and their girlfriends Silda and Mihaela. We had so much fun together. I am also thankful to Informatics Football for providing me with the opportunity to play football in Edinburgh and for Michail to make it even more fun by joining me.

Finally, I want to thank my family. I am very thankful to my wife Rachel for her support during the Ph.D., for always being there for me when I needed a hug, for taking over the household when I was too busy because of various deadlines and for not getting tired of my constant complaining about the Scottish weather. I also want to thank my brother Christian for being my best friend ever since he was born and always being there for me when I need him. This thesis is dedicated to my parents, Manfred and Gisela, because they had the biggest contribution in me reaching this point. They always did everything they could to ensure that I would have the best opportunities in life and, in some way, this thesis is the fruit of their efforts. During my childhood, they supported me with schoolwork, nurtured my interest in science and showed me most of Europe in our little camping car. I will always be grateful for this.

To my parents

Manfred and Gisela

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Fabian Nagel*)

Table of Contents

1 Introduction **1**

2 Background **7**

2.1 Database systems . . . 7

2.1.1 Query processing . . . 7

2.1.2 Query compilation . . . 10

2.1.3 Columnar versus row-wise storage . . . 11

2.2 Managed runtimes . . . 12

2.2.1 System architecture . . . 12

2.2.2 Garbage collection . . . 13

2.3 Language-integrated query in C♯ . . . 15

2.3.1 LINQ-to-objects . . . 16

2.3.2 LINQ query provider . . . 19

2.4 Object-relational mappings . . . 21

2.5 Memory management . . . 24

2.5.1 Safe manual memory management . . . 24

2.5.2 Safe memory reclamation in lock-free data structures . . . 26

2.6 TPC-H . . . 27

3 Query compilation for language-integrated query **29**

3.1 Introduction . . . 29

3.2 Related work . . . 33

3.3 Query compilation architecture in C♯ . . . 33

3.4 Compiling to C♯ code . . . 41

3.4.1 The generated code . . . 41

3.4.2 The code generation process . . . 46

3.5 Staged query processing . . . 48

3.5.1 The generated code . . . 49
3.5.2 The code generation process . . . 57
3.6 Evaluation . . . 61
3.6.1 Experimental setup . . . 61
3.6.2 Compiled C♯ . . . 63
3.6.3 Staged query processing . . . 65
3.7 Summary . . . 71

4 Black-box collection 73
4.1 Introduction . . . 73
4.2 The basic collection type . . . 79
4.2.1 The generated code . . . 81
4.2.2 The code generation process . . . 84
4.3 Columnar storage . . . 85
4.4 Related work . . . 86
4.4.1 Building database systems in high-level languages . . . 86
4.5 Evaluation . . . 88

5 Safe manual memory management 93
5.1 Introduction . . . 93
5.2 Design overview . . . 94
5.3 Safe manual memory management . . . 96
5.3.1 Incarnation number overflow . . . 97
5.3.2 Memory contexts . . . 99
5.4 Concurrency . . . 99
5.4.1 Freeing objects . . . 103
5.4.2 Allocating objects . . . 105
5.5 Concurrent compaction . . . 108

6 Self-managed collections 115
6.1 Introduction . . . 115
6.1.1 Collection semantics . . . 115
6.1.2 Integration into the managed runtime . . . 118
6.2 The basic collection type . . . 118
6.3 Concurrent compaction . . . 122
6.4 Direct pointer . . . 124

6.5 Columnar storage . . . 128
6.6 Evaluation . . . 129
6.6.1 Sensitivity to relocation threshold . . . 130
6.6.2 Evaluating collection primitives . . . 131
6.6.3 Enumeration and query performance . . . 133
6.7 Related work . . . 137
6.7.1 Object-oriented databases . . . 137

7 Conclusion and discussion 141

Bibliography 145

Chapter 1

Introduction

In this thesis, we explore several strategies to improve the performance of query processing inside the memory space of managed applications. In particular, we examine how query processing and storage management technologies from database systems can be applied in the context of managed applications to improve query processing performance.

Over the past two decades, DRAM prices have been dropping at an annual average of 33% with this trend projected to continue. As of July 2015, enterprises can buy servers with a DRAM capacity of more than 1TB for under US$50,000. For many applications, these servers allow the entire working set of the application to fit into main memory. This trend has led to the emergence of in-memory database systems (IMDBs). By storing and processing all data in main memory, these systems outperform traditional database systems that are optimized for disk access. In-memory database systems typically serve applications written in an object-oriented language running on a managed runtime. For example, a C♯ application or an ASP.NET application running on a web server that utilize an IMDB as their data back-end.

Such deployments have to deal with the impedance mismatch between the data models of the object-oriented application and the relational database. To address this, the application developer has to think at two different levels. The application level, where data is represented in the object-oriented data model of the host programming language, and the data processing and manipulation level where data is typically expressed in the relational model and processed through SQL queries. The programmer has to either take care of the impedance mismatch through manual and potentially error-prone data model translation, or through a high-level object-relational mapping that bridges the two data models and manipulation methods such as LINQ-to-SQL. We

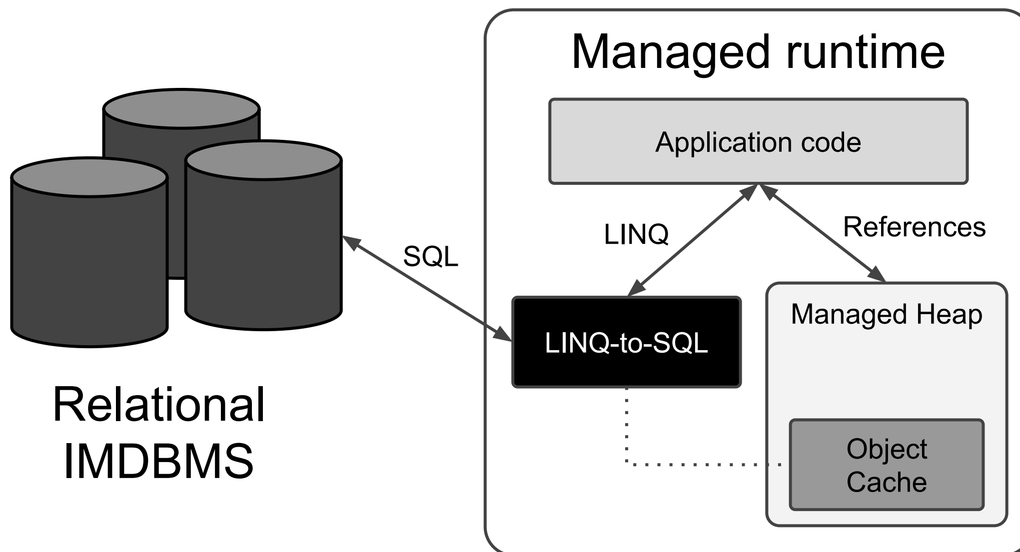


Figure 1.1: Data managed in external IMDBMS

illustrate this scenario for a C♯ application that uses LINQ-to-SQL to communicate with an external database system in Figure 1.1.

In many cases, these applications run on the same server instance as the database system. This is, for example, the case for analytics and business intelligence applications where almost all of the application processing is spent inside the database system and the application is merely responsible for facilitating the interaction with the database system by providing a graphical user interface and visualizing query results (e.g., as graphs, diagrams or tables). These applications usually have a very low number of concurrent users and often only perform off-line modifications to the data stored in the database. Furthermore, they tend to aggregate huge volumes of data into a few summarizing records that are then presented to the user through GUI elements.

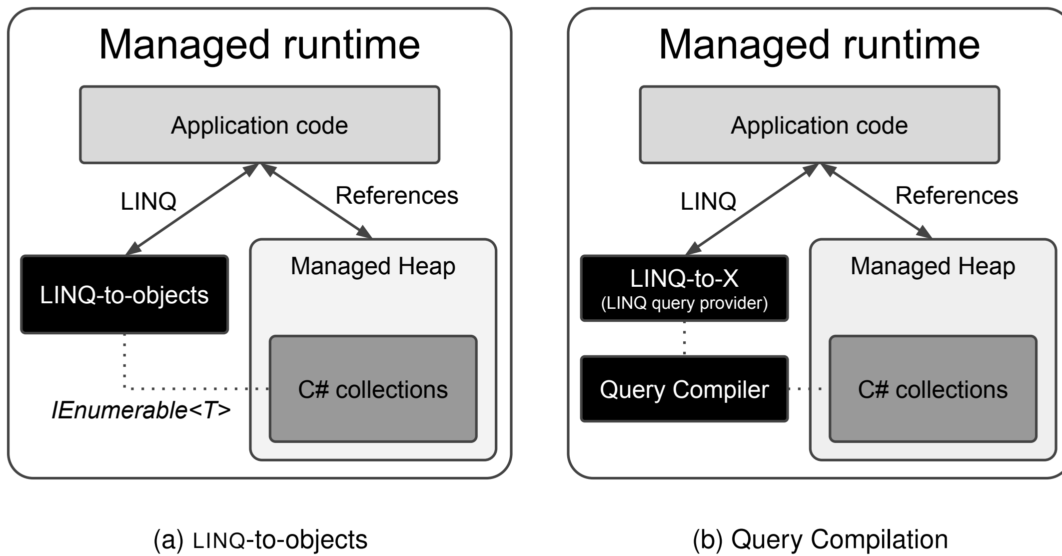
These applications often do not need the full range of functionalities that the database system is capable of. For instance, applications that do not perform on-line modifications to the data set do not need support for ACID transactions. Even with on-line modifications, the application might not require the most recent view of the data as the summarizing results produced by analytics queries are not hugely affected by small changes to the data set and these application often are more interested in analyzing past data (i.e., past days, months, years) than the most recent changes. As a result, these applications tend to have low consistency requirements.

We argue that such applications could instead store and process their data in the memory space of the application, e.g., as collections of objects, rather than employing

an external database system. Doing so has several advantages. It enables a deeper integration of the data store into the application and the use of the full expressive power of the object-oriented programming language to access, modify or query data. For instance, application logic could directly access and modify data elements without the need to marshal data between different systems or extend query processing beyond what is supported in SQL (e.g., add new statistical query processing primitives or application-specific logic). Modeling data using object-oriented principles, further, allows a nested representation of the data sets where related objects are linked through references. Following references between related objects can improve the query performance compared to the corresponding relational representation as the latter has to perform more expensive foreign-primary-key joins to access related objects. In addition, this approach eliminates the impedance mismatch problem as relational database systems are not longer involved.

However, managing and processing data in the application is far from straightforward and often results in degraded performance compared to databases. Traditionally, the developer has to implement all queries using imperative code, which pollutes the code base and relies on complex optimizations being applied by hand, sacrificing maintainability and extensibility in the process. These problems have led to the introduction of language integrated query which adds query capabilities to the syntax of the host programming language; in particular, most implementations add general-purpose query operators to the programming language, e.g., C♯ with LINQ [Meijer et al., 2006] and Java with (parallel) Streams. These operators resemble relational operators. They are concatenated to compose more complex queries at ease. Language-integrated query successfully addresses the impedance mismatch, as all queries are specified in the syntax of the host programming language. In the following chapters, we will focus on C♯ as the programming language and LINQ as the integrated query language, however, most of our findings can be applied to other languages as well.

This thesis will present several strategies to improve query processing in the memory space of the application. We take LINQ-to-objects, C♯'s implementation for language-integrated query on collections of objects, as the starting point. We illustrate the architecture for processing objects stored in the managed heap through LINQ-to-objects in Figure 1.2a. In Section 2.3, we first look at the execution model of LINQ-to-objects where we find parallels to volcano-style iterators [Graefe, 1994], an execution model that has traditionally been used in relational database systems and exhibits a high interpretation overhead. But we also find other inefficiencies that query processing in



(a) LINQ-to-objects (b) Query Compilation

Figure 1.2: Data managed in managed heap

managed runtimes suffer from, most prominently, the cost of automatic garbage collection.

Based on the parallels between the execution model of LINQ-to-objects and that of traditional relational database systems, we address the inefficiencies of the former by leveraging query compilation, an execution strategy that has been introduced in the database space [Krikellas et al., 2010, Neumann, 2011] to address the inefficiencies of volcano-style iterators. Query compilation refers to dynamically compiling SQL query statements into specialized low-level code that executes the query on the data store of the database system. The generated code is then compiled and executed to produce the result of the query. In the database space, this execution strategy has been shown to greatly improve query performance. In Chapter 3, we apply query compilation to query processing using language-integrated query on collections of objects stored in the memory space of the application. In Figure 1.2b, we illustrate the overall architecture of this approach where LINQ-to-objects is replaced by a custom query compiler. We then contribute an alternative query evaluation strategy that further improves query performance by staging object-oriented data into temporal buffers and then using `unsafe` C♯ or native C code to process the query.

We identify the object-oriented data layout and the impact that automatic garbage collection has on it as another bottleneck for query performance. To address it, we propose to take garbage collection out of the equation by storing data that is predominantly used for query processing through LINQ in *unmanaged* memory. In addition

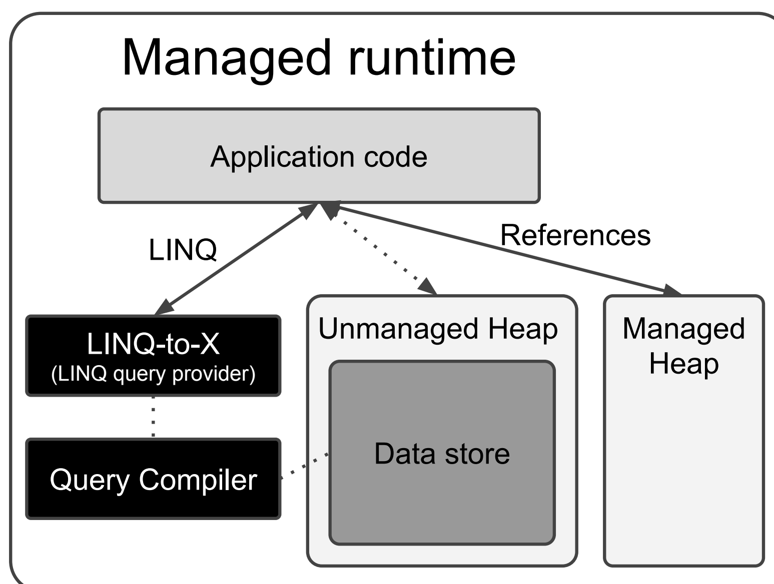


Figure 1.3: Data managed by DBMS-inspired runtime

to the managed heap that contains all traditional object-oriented data the application utilizes a second, unmanaged heap. We illustrate this scenario in Figure 1.3. In Chapter 4 we introduce black-box collections, a collection type that internally stores data in the unmanaged heap similar to the in-memory data store of relational database systems and utilizes existing object-relational mapping schemes to provide the illusion of the collections containing object-oriented data. We adapt our query compiler to generate code that directly processes queries on the unmanged database-like data store of black-box collections. Evaluating the query performance of black-box collections showcases huge improvements compared to LINQ-to-objects, in some cases over an order of magnitude.

However, the performance improvement of black box collections is achieved at the cost of generality. Data stored in the collections is no longer object-oriented, i.e., it does not support references, and the object-relational mapping used to provide an illusion of the collections containing objects causes overheads on the system and requires additional efforts by the programmer. To improve on this, we introduce self-managed collections, a novel collection type, that is designed to provide fast query performance while allowing to store data as objects. Self-managed collections are supported by a novel type-safe manual memory system that manages all collection objects outside garbage collection and by collection semantics that differ from those of regular managed collections which allows self-managed collections to automatically manage con-

tained objects. We will present the manual memory management system in Chapter 5 and self-managed collections in Chapter 6. Self-managed collections further improve query performance compared to black-box collections.

The remainder of this thesis is structured as follows. Chapter 2 will give an overview of existing query processing techniques in traditional database systems and in C$^\sharp$. Chapter 3 will present an architecture for query compilation in C$^\sharp$ and techniques to improve the performance of compiled queries. Chapter 4 will present black-box collections, a collection type that utilizes the in-memory data store of a database system to improve query processing performance. Chapter 5 will describe the implementation details of a type-safe manual memory management system that is purpose-built for self-managed collection, a collection type that delivers comparable query performance to black-box collections, but allows for a deeper integration into the application by storing manually managed objects instead of relational records. Self-managed collections will be presented in Chapter 6. Finally, Chapter 7 will conclude the thesis and outline some future work directions.

Chapter 3 and Chapter 4 have been published in *Proceedings of the VLDB Endowment* [Nagel et al., 2014].

Chapter 2

Background

This chapter provides background information on several topics that the following chapters build on. In Section 2.1, we first outline query processing techniques in traditional database systems and then focus on query compilation, a query processing technique that gained traction in recent years. In Section 2.2, we provide a brief overview of the architecture of managed runtimes and describe garbage collection in more detail. In Section 2.3, we look into query processing in C♯. We describe the architecture of LINQ when querying managed objects in the memory space of the application and the architecture of LINQ query providers. The latter allow LINQ to query custom data sources. In Section 2.4, we outline object-relational mappings by using LINQ-to-SQL as an example. Finally, we look into manual memory management techniques in Section 2.5 and the TPC-H database benchmark in Section 2.6.

2.1 Database systems

2.1.1 Query processing

Relational database management systems (DBMSs) typically evaluate complex SQL queries by concatenating multiple query operators, each representing a relational operation (e.g., selection or projection). Traditional relational DBMSs use the *volcano iterator model* [Graefe, 1994] to evaluate the resulting query tree on top of their data store. Volcano iterators define a common interface (or base class) that all query operators implement. It contains an `open()` method that initializes the operator state before query evaluation, a `close()` method that cleans up the operator state after query evaluation and a `next()` method that is called by its parent operator to ask for (a pointer to)

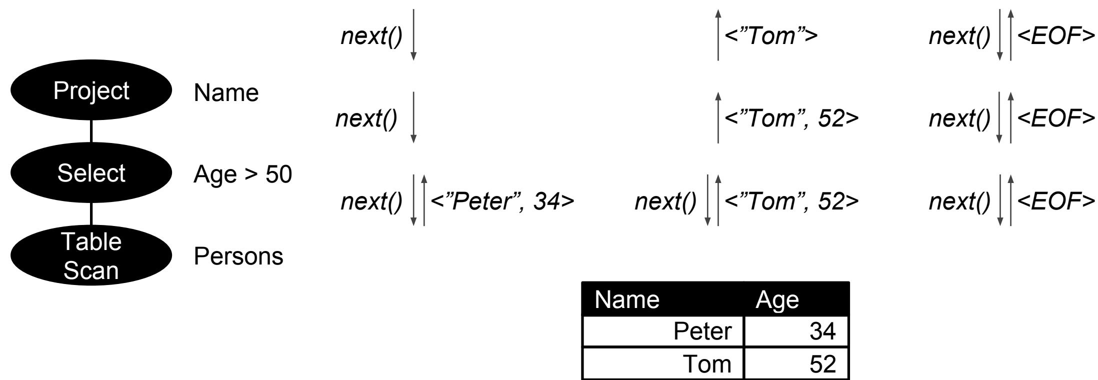


Figure 2.1: Query evaluation in the volcano iterator model

the next element of its result. Query evaluation is driven by issuing an `open()`, (multiple) `next()` and finally a `close()` call on the root operator of the operator tree. To produce its output, the root operator has to in turn issue `open()`, `next()` and `close()` calls to its child operator(s). This process is repeated until a leaf node is reached, which represents a table scan operator. The data flow within the operator tree is pull based as operators pull data upwards. In Figure 2.1, we illustrate the evaluation of a simple query that returns the names of all persons with an age greater than 50 using the volcano iterator model. Note that the select operator issues several `next()` calls to its child operator before it can produce its first output element. The classic execution paradigm that is used with the volcano iterator model is referred to as *tuple-at-a-time* as data is propagated through the operator tree at the granularity of a single tuple.

Query evaluation in database systems is traditionally interpretative. At database compile time, there is no knowledge about the schemas that the database system will host or the queries that will be evaluated on it. As such, query performance is heavily dependent on the overhead that is imposed by the interpretative approach. This overhead includes the cost of calling volcano's `open()`, `next()` and `close()` functions. As the operator tree is only constructed at runtime based on user input (the SQL statement), the implementation of the query operators that implement the volcano iterator interface is not aware of their parent or child operators in the operator tree. At compile time, the target of the `open()`, `next()` and `close()` functions is not known and, therefore, the compiler cannot inline them, thereby increasing the cost of calling these functions, while also preventing otherwise possible compiler and processor optimizations. The implementation of each query operator is also unaware of its input. This includes the number of columns, the types of each column and, in some cases, the specific expressions that need to be evaluated on the input (e.g., selection

conditions). The operator state stores meta data specifying these parameters and each `next()` call has to interpret its input using the meta data and a multitude of conditional statements and additional function calls. Tuple-at-a-time, the classic execution paradigm of the volcano iterator model exacerbates these overheads as they have to be amortized over a single tuple [Boncz et al., 2005]. This, further prevents compiler optimizations such as loop unrolling, loop pipelining, strength-reduction and automatic SIMDization [Żukowski, 2009]. As a result, the execution paradigm does not expose enough (instruction-level) parallelism to keep modern out-of-order superscalar CPUs busy [Ailamaki et al., 1999].

Research addressed this issue by extending the volcano iterator model or proposing new execution paradigms. In the operator-at-a-time paradigm [Boncz, 2002], each operator processes its entire input at once and fully materializes its output to be consumed by the consecutive operator. This execution paradigm reduces the interpretation overhead (e.g., virtual function call overheads) as the overhead is amortized over all input tuples. It, further, allows loop-based compiler optimization and exposes more (instruction-level) parallelism to the CPU. However, the paradigm comes at the cost of having to materialize all intermediate results during query processing and of missed processing opportunities while a tuple resides in CPU caches as tuples that are loaded into CPU caches are only processed by a single operator before being flushed out to make space for the remaining input tuples. Other research addresses these issues by proposing block oriented processing [Padmanabhan et al., 2001] or the vector-at-a-time paradigm [Boncz et al., 2005]. The latter maintains the volcano iterator model, but propagates data through the operator tree at the granularity of a fixed-size vector (e.g., 1024 tuples). All interpretation overhead is amortized over the size of a vector. The vector size is chosen to ensure that all vectors that are processed by the operator pipeline at the same time all fit in the CPU cache.

In recent years, research has gone one step further by suggesting to scrap interpretative query evaluation all together and instead dynamically compile SQL queries to imperative and highly specialized native code [Krikellas et al., 2010, Neumann, 2011, Rao et al., 2006] to eliminate all interpretation overhead, allow advanced compiler optimization and improved CPU and cache utilization.

2.1.2 Query compilation

A database system utilizing query compilation operates similar to a traditional, interpretative database system. Incoming SQL statements are parsed into a tree representation of the query. Each node of the tree specifies either a relational operation or parameters of a relational operation. The optimizer rewrites this query tree into a optimized query tree based on the database schema, collected statistics (e.g., histograms) and a cost model. Instead of building and executing an operator tree from the query tree, query compilation based systems emit native (e.g., C or LLVM) code to evaluate the query. The generated code is then compiled and executed. As the code is compiled at run time where all characteristics about the query and the underlying database schema are known, the query compiler can generate specialized query code that does not require any interpretation. Further, queries are evaluated in tight loops over the input that allow advanced compiler optimization and improved CPU and cache utilization. In contrast to volcano iterators, query compilation represents a push-based approach where data is pushed through several operators inside a loop construct. However, research [Krikellas et al., 2010] has also noted the high cost of compiling queries to binary code, in particular if compilation is to be performed using an external compiler process.

Query compilation in database systems has been around since the times of System-R [Chamberlin et al., 1981]. However, as its query compiler directly generated assembly, it was abandoned for its lack of portability. [Freytag and Goodman, 1989] suggested to evaluate SQL queries by generating C or Pascal code, compile it using the existing compiler infrastructure and execute it. The code generator uses functional programming and program transformation techniques to translate optimized query plans into iterative programs. It translates the query plan into map expressions that are modelled after the Lisp operator `map` and lambda expressions. Multiple transformations (e.g., loop fusion) are then applied to generate iterative code. Performance aspects were only a minor concern. At the time, query performance in database systems was disk I/O bound and improvements to CPU or cache efficiency only had a minor impact. Since then, supported by a constant increase in main memory capacity, memory performance became the new bottleneck and processing queries in a CPU and cache efficient manner became imperative. As a result, query compilation has gained renewed interest in the research community in recent years which let to multiple industrial applications such as SQL Server's Hekaton engine [Diaconu et al., 2013], Amazon Red-

shift [Gupta et al., 2015], Cloudera Impala [Wanderman-Milne and Li, 2014] or IBM Netezza [Francisco et al., 2011].

[Rao et al., 2006] 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 that only contain primitive types (i.e., no references). [Krikellas et al., 2010] propose generating C code instead. The authors precede query processing with a staging phase that prepares the input data for cache conscious query processing. The authors also note the high cost of compiling the generated C code. [Neumann, 2011] 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 for as long as possible. [Klonatos et al., 2014] propose the use of a high-level programming language to implement database systems in order to increase the productivity of database engineers. The authors propose to use query compilation to enable fast query processing; in particular, their query compiler performs a source-to-source compilation of high-level Scala code to low-level C code that is then compiled and executed. A multitude of other work has looked into various aspects of query compilation, e.g., [Sompolski et al., 2011] studied query compilation at an operator level, [Dees and Sanders, 2013] explored code generation for many-cores in main memory column-stores and [Pirk et al., 2013] looked into the symbiosis of query compilation and the partially decomposed storage model.

2.1.3 Columnar versus row-wise storage

The records in database tables are usually either stored columnar or row-wise. Row-wise layouts store the fields of a record (columns) in consecutive memory addresses. In a columnar data layout, the storage of each database table is vertically decomposed into memory areas that each only contain the values of a single column (i.e., a single field). Vertically decomposing the storage of a table can lead to better query performance as it enables an improved utilization of CPU caches and prefetching, e.g., [Copeland and Khoshafian, 1985, Manegold et al., 2000]. Columns that are not accessed by the query are never fetched into CPU caches and a single cache line fetch of a column that is accessed by the query reads many consecutive values of that column into the CPU caches. As a result, queries that either do not touch a significant fraction of a table's columns or that tend to touch consecutive column values perform better when using columnar storage. However, columnar storage comes at a higher CPU cost when

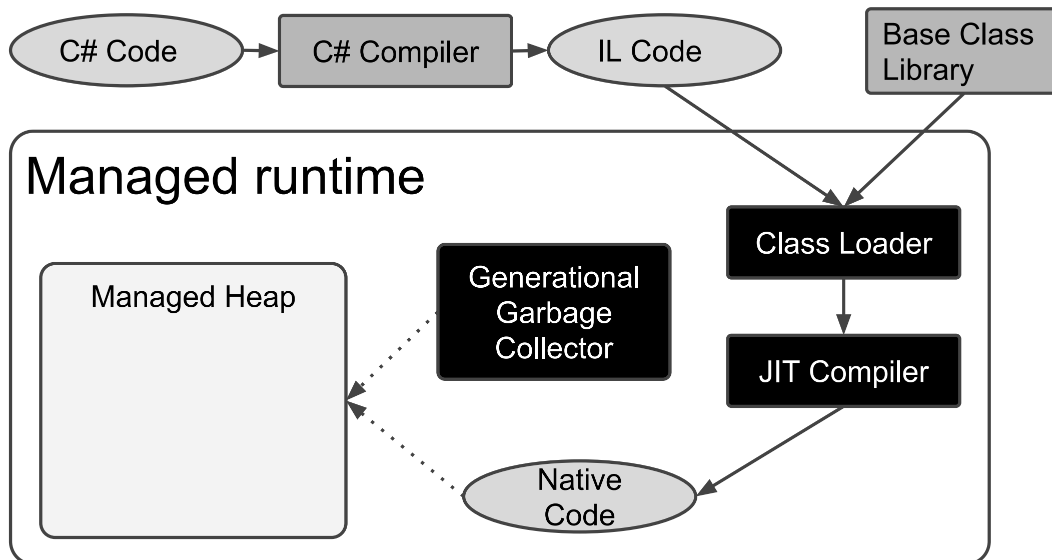


Figure 2.2: Overviews of .NET's common language runtime (CLR)

accessing the columnar data as memory offsets have to be computed for each value. An additional strength of columnar storage is that it works very well with compression schemes [Abadi et al., 2009] because all data stored in a column has the same type and often has a rather small value domain or small increments between successive values (e.g., primary key columns). PAX (Partition Attributes Across) [Ailamaki et al., 2001] is an alternative data layout that, instead of vertically partitioning the table as is the case for columnar storage, only vertically partitions the records within a memory block. This enables similar memory access properties as columnar storage while maintaining the property of row-wise storage that records are stored in a single block.

2.2 Managed runtimes

2.2.1 System architecture

Applications written in many modern object-oriented programming languages require a managed runtime to be executed; examples include the *Common Language Runtime* (CLR) which is shared between various .NET languages (e.g., C♯ or Visual Basic) or the *Java Runtime Environment* (JRE). In Figure 2.2, we give a high-level overview of the CLR and the process that a C♯ application undergoes when being compiled and executed. The C♯ compiler compiles the application's source code into intermediate language (IL) code. When running the application, each IL method is compiled into

native machine code by the just-in-time (JIT) compiler when it is executed for the first time. Later invocations of the same method directly execute the compiled native code. All objects created by the application are allocated from the managed heap which is managed by the garbage collector. Hitting certain memory limits when allocating new objects triggers garbage collections that automatically free the memory space of stale objects. Parts of the garbage collection process require all other application threads to be suspended.

2.2.2 Garbage collection

Most modern managed runtimes employ a generational garbage collector [Ungar, 1984, Lieberman and Hewitt, 1983]. Generational refers to the fact that the managed heap is divided into several generations, usually two or three. All objects created by the application are allocated in the youngest generation and then gradually moved to older generations by garbage collections as they mature. Allocating memory in the youngest generation is typically very cheap as the memory space is thread-private and most allocations can be satisfied by pointer pushing. Once the youngest generation is full, a garbage collection is invoked that copies all live objects, i.e., objects that are still reachable by the application, to the next mature generation. This process repeats for more mature generations. Typical implementations allow to collect the memory space of a specific generation individually where collections in younger generations are usually much cheaper. After collecting the oldest generation, objects can no longer be moved to older generations and instead remain in the same generation. The holes left by unreachable objects are either filled by performing a compaction step at the end of the garbage collection that moves reachable objects together or by maintaining free lists that are filled in following garbage collections when moving objects from the next younger generation. In the latter case, unused memory is managed as lists of memory blocks catering for a specific range of object sizes. Each memory block is organized by memory slots of the maximum size of that range and contains a list of free object slots to be used for allocations. Both options prevent fragmentation and reduce the overall memory footprint. Generational garbage collection typically improves garbage collection performance by exploiting a common allocation pattern in managed application. The more recently created objects are also the ones that are more likely to become unreachable. In particular, this implies that most objects tend to be short-lived and, therefore, are not likely to survive a collection in the youngest generation. As alloca-

tions and garbage collections in the youngest generation are significantly cheaper than in more mature ones, generational garbage collection employs frequent collections in the youngest generation to get rid of short-lived objects and only performs the more expensive collections in more mature generations very rarely.

Most garbage collectors are based on mark-and-sweep collectors [McCarthy, 1960] which consist of two phases, the marking phase that finds all objects that are still reachable by the application and the sweep phase that goes through all memory blocks in the managed heap and reclaims the memory of all objects that have been deemed unreachable. All objects that are reachable in the application form a graph from the application roots where objects represent nodes in the graph and object references the edges. Application roots are objects that are referenced from the call stack (e.g., local variables or function parameters) or from global variables. The marking phase starts with the application roots and performs a depth first search that follows all their references and marks the memory slots of all reached objects. In generational garbage collectors, the runtime tracks references that cross generation boundaries. This allows garbage collections in these generations to limit the marking phase to objects stored in the memory space of a certain generation by treating objects referenced from more mature generations as additional application roots and by stopping to follow references beyond generation boundaries. Doing so allows to perform garbage collections in the memory space of individual generations, without having to inspect objects in other generations, and, hence, to significantly reduces the cost of such collections. After the marking phase, the sweeping phase iterates over all memory slots in all memory blocks in the generation to be collected and reclaims the memory slots of all objects that have not been marked, as well as resetting all marked objects in preparation for future garbage collections.

Blocking garbage collectors have to suspend all application processing while garbage collections are in progress. Modern garbage collectors also support concurrent modes where application processing only needs to be halted for some parts of the collection process, e.g., for scanning the application's stack. As a result, concurrent garage collections reduce the maximum response time of the application, however, in many cases it also increases the total time spent on garbage collection compared to batch collections.

.NET's garbage collector is a generational mark-and-sweep garbage collector that divides the managed heap into three generations (and an area for large objects). It supports two modes, workstation and server, that specify whether the garbage collection

is performed by the application thread that triggered it (workstation) or by dedicated garbage collection threads, one per CPU (server). Both modes support concurrent (interactive) or non-concurrent (batch) collections. In both cases, collections of the two youngest generations suspend all application processing for the duration of the collection. However, concurrent collections of the most mature generation only suspend application processing for small parts of the collection duration.

2.3 Language-integrated query in C♯

Language-Integrated Query (LINQ) is a framework introduced by Microsoft that adds powerful query-like capabilities to C♯ and other .NET programming languages. This is achieved by defining a design pattern of general-purpose query operators that are combined to form more complex queries and by extending the programming languages with special query syntax that is compiled into these operators. The LINQ framework also provides a number of domain-specific implementations of these query operators which enables the 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 p in Persons
               where p.Age > 50
               select p.Name;
```

The query specifies to return the names of all person objects in a Persons collection that are older than 50. This query syntax is merely convenient syntactic sugar, as it is compiled away to a series of method calls, e.g.:

```
var qry_stmt = Persons
              .Where(p => p.Age > 50)
              .Select(p => p.Name);
```

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., p => p.Age > 50) as arguments, and some of these methods directly correspond to relational algebraic operations. They are concatenated to compose more complex query 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.

2.3.1 LINQ-to-objects

For types that implement the IEnumerable or IEnumerable<T> interfaces, e.g., managed .NET collections such as Array or List<T>, the base class library provides an implementation of the standard query operators. This implementation is known as LINQ-to-objects. In this section, we will have a closer look at the inner workings of LINQ-to-objects.

Each of the standard query operators is implemented as an iterator method as illustrated for the Where and Select operators in Figure 2.3. Iterator methods are again syntactic sugar to facilitate the implementation of the standard iterator pattern. From the iterator method, the C$^\sharp$ compiler automatically generates a class implementing IEnumerator<T>, the actual iterator, and IEnumerable<T>. The compiler also modifies the iterator method to merely create a new instance of the generated class and return it to the caller as an IEnumerable<T>. In Figure 2.4, we illustrate a heavily simplified version of the code generated by the compiler for the Where iterator method from Figure 2.3a. Note that the arguments of the original iterator method are passed to the iterator.

Consider the following LINQ query in method syntax:

```
var qry_stmt = Persons
              .Where(p => p.Age > 50)
              .Select(p => p.Name);
```

```
1 IEnumerable<Tin> Where<Tin>( this IEnumerable<Tin> input,
2                              Func<Tin, bool> predicate)
3 {
4   foreach (Tin elem in input) {
5     if (predicate(elem))
6           yield return elem;
7   }
8   yield break;
9 }
```

(a) Where

```
1 IEnumerable<Tout> Select<Tin, Tout> ( this IEnumerable<Tin> input,
2                                         Func<Tin, Tout> selector)
3 {
4   foreach (Tin elem in input)
5         yield return selector(elem);
6   yield break;
7 }
```

(b) Select

Figure 2.3: Implementation of standard query operators

```
foreach (var name in qry_stmt)
  Console.WriteLine(name);
```

When calling the `Where` and `Select` methods in the query statement, the query is not actually executed. Instead, a tree of iterators is constructed. The `Where` method takes the `Persons` collection (as `IEnumerable<Person>`) and a function delegate created from the lambda expression `p => p.Age > 50` as arguments, passes them to the constructor of its iterator and then returns the iterator as `IEnumerable<Person>`. In turn, the `Select` method takes the `Where` iterator (i.e., the `IEnumerable<Person>` returned by the `Where` method) and a function delegate as argument. The `IEnumerable<string>` returned by the `Select` method is then assigned to the `qry_stmt` variable.

The query is only evaluated when the application consumes its result elements (here: in the `foreach` loop). The C♯ compiler translates the `foreach` loop into code

```
1 public IEnumerable<T> Where<T>( this IEnumerable<T> input,
2                                 Func<T, bool> predicate) {
3   return new WhereIterator<T>(input, predicate);
4 }
5
6 internal sealed class WhereIterator<T> : IEnumerable<T>, IEnumerator<T>
7 {
8   private IEnumerable<T> input;
9   private Func<T, bool> predicate;
10  /* ... */
11
12  public WhereIterator(IEnumerable<T> input, Func<T, bool> predicate) {
13    this.input = input;
14    this.predicate = predicate;
15  }
16
17  public IEnumerator<T> GetEnumerator() { return this; }
18  public T Current { /* ... */ }
19  public bool MoveNext() { /* ... */ }
20  public void Dispose() { /* ... */ }
21 }
```

Figure 2.4: Compiler-generated `Where` iterator

```
1 IEnumerator<string> iterator = qry_stmt.GetEnumerator();
2 try {
3   while (iterator.MoveNext()) {
4     string name = iterator.Current;
5     Console.WriteLine(name);
6 } }
7 finally {
8   iterator.Dispose();
9 }
```

Figure 2.5: Compiler-generated iteration

similar to that shown in Figure 2.5. The generated code first assigns the root of the iterator tree, the `Select` iterator, to its `iterator` variable (line 1). It then iterates over the result elements of the `Select` iterator by continuously calling its `MoveNext` method (line 3) and printing the result element returned by its `Current` getter (lines 4 and 5).

The C♯ compiler transforms the `foreach` loops in the query operators to iterate over their input elements (see Figure 2.3) accordingly. To produce its result elements, the `Select` iterator iterates over the result elements of the `Where` iterator which, in turn, produces its result elements by iterating over the elements of the `Persons` collection. Note how this evaluation strategy evaluates the query by pulling objects through the iterator tree, one object at a time. Therefore, the execution model of LINQ-to-objects closely resembles that of volcano-style iterators (tuple-at-a-time) in relational database systems.

One of the characteristics of LINQ is that the evaluation of a query is deferred until the moment that a result element is actually requested by application code. In our example, the definition of the query does not execute it, instead, iterating over the result elements of the query in a `foreach` loop produces one result element at a time as requested by each loop iteration. For instance, adding a break statement that exits the loop once ten result elements are produced stops the query evaluation after the tenth element. The deferred execution strategy allows to dynamically construct new queries by combining existing query statements, but also reduces the query evaluation cost if the application does not consume the entire query result.

2.3.2 LINQ query provider

The implementations of the LINQ standard query operators are overloaded. The base class library also provides implementations for two additional interfaces which are derived from the enumerable interfaces: `IQueryable` and `IQueryable<T>`. These implementations of the standard query operators provide the means by which data-source-specific implementations, e.g., LINQ-to-SQL or LINQ-to-XML, are defined. Such implementations are known as LINQ query providers.

The chief difference between the LINQ-to-objects implementation of the standard query operators is that executing the query statement builds an expression tree instead of an iterator tree. An expression tree is an AST representation of a given query. Any lambda expression arguments of query operators are quoted, i.e., they are implicitly converted into expression trees. This implicit conversion is implemented by the C♯ compiler. After calling the methods defined in the query statements, the `qry_stmt` variable holds an `IQueryable<string>` object that contains the expression tree. The expression tree is made up of expression nodes of various types. The following three are of particular interest to us:

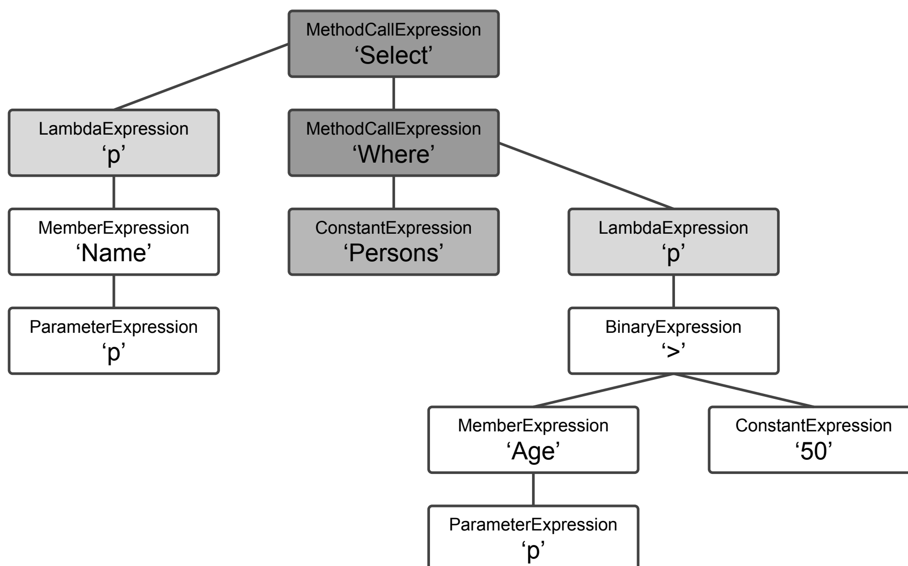


Figure 2.6: Expression tree of the example query

- MethodCallExpression: This node type represents the standard query operators and, hence, the operation that has to be performed on the data returned from its children (which are also MethodCallExpression or ConstantExpression nodes)

- ConstantExpression: This node type represents constants; in particular, the input collections.

- LambdaExpression: This node and its sub-tree specifies the lambda expression arguments of the query operators. As such, they detail the variable parts of the operations to be performed by a standard query operator.

We illustrate the expression tree that is constructed for our sample query in Figure 2.6. The expression tree nodes also contain type information, which is omitted in the illustration. In the case of MethodCallExpression nodes, the type information includes their output types (e.g., IQueryable<Person> for the Where node). Note the resemblance of expression trees to query trees used in database systems to represent and optimize SQL queries.

Other than storing a reference to the expression tree, the IQueryable<string> object that is assigned to the qry_stmt variable also holds a reference to the query provider of the data source. When the evaluation of a query is triggered, e.g., by a foreach loop iterating over the query result, the GetEnumerator method is called on the IQueryable<string> object which is assigned to the qry_stmt variable. The

implementation of the `GetEnumerator` method invokes the query provider implementation to supply the `IEnumerator<string>` requested by the loop; the expression tree is supplied as argument. It is now up to the query provider implementation to return an iterator that is capable of evaluating the query on the data source. Typical implementations interpret the expression tree, transform it into a representation of the native syntax of the data source and then execute the query on the data source. For instance, the LINQ-to-SQL query provider translates the expression tree to a SQL query statement which is then executed on the external DBMS. The iterator finally creates C♯ objects from the result elements returned by the data source once they are requested by the `foreach` loop. We provide additional details on LINQ query providers in Section 3.3.

2.4 Object-relational mappings

When accessing database systems from an object-oriented programming language, there is an impedance mismatch between both data representations. This mismatch is typically addressed by employing an object-relational mapping framework to automatically map between both representations to save the application programmer the effort of manually having to deal with the mismatch. To facilitate the integration of a relational database system into C♯ applications, .NET provides the LINQ-to-SQL query provider and object-relational mapping framework. It manages the connection with the external DBMS as well as the representation mismatch. The latter is addressed by allowing the application programmer, once the mappings are created, to access data stored in the external DBMS as if it was managed objects, stored in an in-memory collection. LINQ provides the means to formulate complex queries to the database system.

There is a mismatch in how data is represented in object-oriented programming languages and relational databases. Data in object-oriented languages is represented by objects that form graphs, connected by references, to represent relationships between them. Identity between two object references is usually defined by both referring to the same instance of a type. The lifetime of objects is defined based on whether there are references in the application through which the object can still be reached. It is not necessarily linked to the object's containment in a collection. In relational databases, on the other hand, data is defined in terms of the relational model. Data elements are represented as table rows. Relationships are loosely specified by primary and foreign keys and related data elements are accessed together through explicit join operations. Identity is defined by the equality of primary keys. The lifetime of data elements

```
1  [Table(Name="Orders")]
2  public class Order
3  {
4     [Column(IsPrimaryKey=true, IsDbGenerated=true)]
5     public int OrderID;
6
7     [Column(CanBeNull=false)]
8     private int CustomerID;
9
10    private EntityRef<Customer> _Customer;
11
12    [Association(IsForeignKey = true, Storage="_Customer",
13                 ThisKey="CustomerID")]
14    public Customer Customer {
15       get { return this._Customer.Entity; }
16       set { this._Customer.Entity = value; }
17    }
18 }
```

Figure 2.7: Annotating a class definition to allow it to be mapped to the data stored in a relational DBMS

stored in a database table is defined by their containment in the table. Removing them from the table erases the data elements and explicitly (e.g., indexes) or implicitly (e.g., foreign-key relationships) removes all references to it.

LINQ-to-SQL allows to enrich the class definition of managed classes that refer to tables stored in database systems by adding attributes (in squared brackets) to the class definition. These attributes allow the runtime to create the mapping between the object-oriented type used in the managed environment and the one defined in the schema of the database system. We illustrate a LINQ-to-SQL class definition that is annotated with attributes to allow it to be mapped to data stored in a database system in Figure 2.7. The `Table` attribute (line 1) specifies that the class type is backed by a table in a relational database and the table's name in the database. The `Column` attribute (lines 4 and 7) specifies all object fields that are backed by the database and properties of the corresponding columns, for example, whether the column is part of a primary key or may contain `null` values. To form relationships (here, with a `Customer` object), the class definition specifies an `Association` attribute (line 12). The `Storage` parameter specifies the name of the object's `EntityRef<Customer>` field that will hold the instance

```
1 public class DBContext : DataContext
2 {
3   public Table<Customer> Customers;
4   public Table<Order> Orders;
5   public DBContext(string connection): base(connection) {}
6 }
```

Figure 2.8: Example of the `DataContext` class

of the other class and `ThisKey` the name of the object's field that holds the foreign key to the other table. The `Association` attribute allows to access related objects in an object-oriented manner using the dot notation (e.g., `order.Customer.Name`). When evaluating the query on the database system, these object accesses are automatically translated into relational joins. When retrieving objects in a query, LINQ-to-SQL uses deferred loading to only retrieve related objects when attempting to access them. For instance, when retrieving orders from the database, related customer objects are only retrieved once they are accessed by the application.

A `DataContext` represents the glue between database and application. Its definition specifies `Table<T>` collections for all accessible tables in the database (as shown in Figure 2.8). As identity has different meanings in the object-oriented and the relational world, the `DataContext` translates between both. Identity in a database means equivalent primary key(s) whereas identity in an object-oriented language is defined by the same object instance. To guarantee the latter, when retrieving the same row (row with same primary keys) multiple times, the `DataContext` has to ensure that also the same object instance is retrieved. For this purpose, the `DataContext` contains an *identity cache*. The identity cache can be seen as a hash table that stores the primary key(s) of a retrieved row together with the reference of the object representing that row. Before returning a row that has been retrieved from the database to the application, the `DataContext` first checks the identity cache if an object for this row already exists and then returns that object, otherwise it creates a new object and adds it to the cache. In the former case, the object is not updated with the current values of the row. Identity is not only important to provide the application with familiar semantics, but also because it provides the means to write back changes that have been performed on objects returned from the database. For this purpose, the `DataContext` tracks all changes that are performed on objects returned from the database.

Note that, as the `DataContext` holds references to all returned objects that represent rows in the database for the cause of its lifetime, these objects cannot be reclaimed by the garbage collector. This causes the `DataContext` to consume huge volumes of memory; in the worst case leading it to contain a second, object-oriented copy of the entire data set stored in the DBMS. For this reason, `DataContext`s are assumed to be short-lived, their lifetime typically is a *unit of work*. `DataContext`s are usually created in a `using` statement to ensure that they are disposed once the unit of work is finished. All changes performed on objects representing rows that have been retrieved from the database are only kept locally until the application explicitly calls the data context's `SubmitChanges` method. The `DataContext` tracks the original values and changes of all objects returned from the DBMS. When `SubmitChanges` is called, all changes are translated into equivalent SQL statements and executed against the database. Note that object identity and, hence, the option to write changes to objects back to the database, is only available in the scope of the data context that retrieved the objects. To reduce the `DataContext`'s memory footprint and to improve performance, object tracking can be turned off in a data context when dealing with a read-only workload.

2.5 Memory management

2.5.1 Safe manual memory management

Automatic garbage collection in managed runtimes typically guarantees memory, type and thread safe memory management for managed types. Memory safety, e.g., ensures that there are no dangling references to already freed memory addresses, no memory leaks or accesses to uninitialized memory. Type safety ensures that references of a certain type always refer to instances of that type. Thread safety ensures that memory management remains memory and type safe, even when facing concurrent operations. Providing safety guarantees without automatic garbage collection is more of a stretch. In the past, several schemes have been proposed to add some safety guarantees to memory operations without utilizing garbage collection. They either improve program safety through static checks at the compiler level, through dynamic runtime checks or both. Reference counting is a common runtime mechanism to ensure memory safety. By counting references to an object or a memory area, these resources can be automatically freed once they are no longer referenced. C++11's `shared_ptr` smart pointers are an example of this mechanism used at object granularity. However, reference

counting comes at a high cost, especially when objects may be accessed concurrently, e.g., [Michael, 2004]. Linear types [Wadler, 1990] are another option to ensure safe memory deallocation by preventing aliases. C++11's `unique_ptr` smart pointers ensure safe deallocations through compile time checks based on linearity.

Region-based memory management has been proposed as an alternative to garbage collection. It allows to allocate objects in regions (e.g., lists of memory blocks) and to efficiently free all objects in a region at once. Memory safety is either added statically or dynamically. [Tofte and Talpin, 1997] maintain a stack of regions and use a program analysis to automatically find program points where entire regions can be allocated and freed and what region to use for an allocation. [Gay and Aiken, 1998] describe a safe implementation of explicit regions, i.e., regions that are explicitly created and freed by the programmer, that uses reference counting to prevent a region from being freed while the reference count is greater zero. [Boyapati et al., 2003] and [DeLine and Fähndrich, 2001] propose static schemes to check the correctness of region management. [Dhurjati et al., 2003] statically ensure type safety of manual allocations and deallocations by automatically assigning heap-allocated objects to type-homogeneous pools (regions) that are destroyed when there are no more references to data elements in a pool. Type safety is ensured by preventing the memory within a pool from being released to the system until the pool is destroyed. Cyclone [Grossman et al., 2002] is a C-like programming language that provides safe manual memory management based on regions.

[Austin et al., 1994] transform programs at compile-time to automatically use an extended pointer representation which they refer to as safe pointers. Safe pointers contain the value of the pointer as well as object attributes like location, size or lifetime. Memory access errors are detected by validating dereferences against the object attributes at runtime. The lifetime of heap objects is validated by storing a unique value that identifies the object instance (memory allocation) in the reference and by checking in a dynamically maintained global table that contains all identifier of object instances that have been allocated but not freed whether the object is still alive. Access is only granted if the object is in the table, therefore, preventing accesses to dangling pointers. Other systems have used similar techniques to dynamically check memory safety. For example, [Dragojević et al., 2014] use fat pointers to track object incarnations to prevent access to objects that have already been freed and to allow to safely reclaim their memory space while there may still be references to them. For this purpose, the memory space that is used to store an object is associated with an incarnation num-

ber and references to the object store the same incarnation number. Freeing the object increments the incarnation number. Access to an object is only granted if the incarnation number stored in the reference matches the one of the object to prevent accessing dangling references.

2.5.2 Safe memory reclamation in lock-free data structures

The manual memory management system that we present in this chapter utilizes techniques that have been proposed for memory reclamation in the context of lock-free data structures. In particular, we use epoch-based memory reclamation to ensure that our memory manager cannot reclaim memory space that is still accessed by a concurrent thread. Lock-free data structures avoid problems like deadlocks that occur when using locking as concurrency control mechanisms and, in some cases, can lead to performance improvements. However, in the absence of automatic garbage collection, they have to handle read/reclaim races. Consider a linked list where a thread T1 removes a node while another thread T2 holds a reference to that node. The memory of the node has to be reclaimed (freed) to allow it to be reused, however, reclaiming it is unsafe while T2 still holds a reference to it.

Hazard pointers and their variants [Herlihy et al., 2005, Michael, 2004] address this problem by introducing a special pointer type, hazard pointers, that protects accesses to elements of lock-free data structures. Each thread maintains N hazard pointers that are globally visible. It uses them to access protected data structure elements. When removing an element, the thread first checks the hazard pointers of all other threads to ensure that the element is only reclaimed if there are no protected references to it. Otherwise it is added to a list to be reclaimed later.

Epoch-based reclamation [Desnoyers et al., 2012, Fraser, 2004] is an alternative approach to ensure safe memory reclamation. Epoch-based reclamation uses time intervals during which threads may safely hold references to protected data structure elements. Removed elements are maintained in lists characterized by the time interval they were removed and, once all threads passed a certain number of time intervals, they are freed safely as there cannot be any more references. We will look at epoch-based reclamation in more detail in Section 5.4 when describing how concurrency is handled in our memory management system. [Braginsky et al., 2013] introduce an epoch-based reclamation scheme that can recover from thread failures by using hazard pointers for every Nth access to protected memory. Their approach also improves the performance

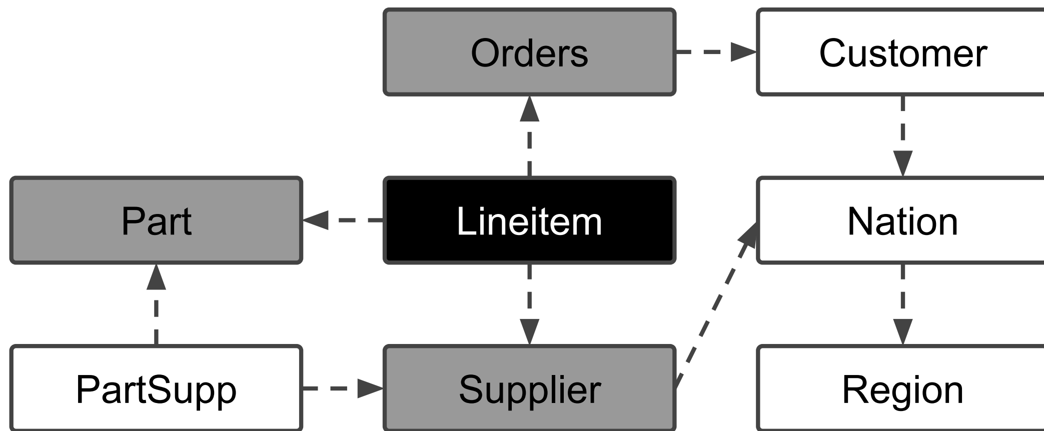


Figure 2.9: Schema of the TPC-H Benchmark

compared to hazard pointers by amortizing the cost of hazard pointers over N accesses to protected memory.

[Braginsky and Petrank, 2011] propose a lock-free sorted linked list optimized for spatial locality by clustering multiple list elements together in a sub-list that is stored in a chunk of consecutive memory addresses within the parent list element. Hazard pointers ensure safe memory reclamation, while a freeze bit in the elements' next pointer ensures lock-free splitting and merging of chunks. The implementation is limited to a specific format for each list element (`integer` key and value).

2.6 TPC-H

TPC-H[1] is a popular business intelligence database benchmark. Its schema contains 8 tables that are arranged in a snowflake scheme with a single fact table (i.e., `lineitem`) and several dimension tables (e.g., `orders` or `supplier`) as depicted in Figure 2.9. In contrast to star schema arrangements, dimension tables in snowflake schemas are normalized forming additional tables (e.g., `customers` or `nations`). The total size of the data sets is specified by its scale factor (e.g., a scale factor of one corresponds to a size of 1GB, a scale factor of two to 2GB, etc.). TPC-H consists of a total of 22 queries that represent typical business intelligence tasks. In the remainder of this work, we focus on the first six queries of the TPC-H benchmark as we believe they are representative of the remaining TPC-H queries as well as other analytics workloads. Query 1 is very aggregation-heavy, query 2 contains a nested sub-query, query 3 to 5

[1] http://www.tpc.org/tpch/

contain several join operations and query 6 is very selection-heavy.

The benchmark itself contains two parts, a *power run* that measures the evaluation time of each of the 22 queries and a throughput run that executes several streams containing the 22 queries together with two refresh streams that modify the data stored in the database. Throughput runs measure the query throughput when evaluating these streams concurrently. One of the two refresh streams removes a predefined percentage of records from the `lineitem` and `orders` tables, the other one inserts the same number of records.

Chapter 3

Query compilation for language-integrated query

3.1 Introduction

When describing LINQ-to-objects in Section 2.3.1, we highlighted the resemblance between its iterator-based execution model and that of the volcano iterator model (tuple-at-a-time) as traditionally utilized in most commercial database systems (Section 2.1.1). Iterators allow complex queries to be composed out of standardized query operators derived from relational operations. They are the means to support LINQ's deferred execution strategy which allows to only produce result elements of a query when they are consumed by the application. However, they also exhibit the same inefficiencies as their counterparts from the database space (described in Section 2.1.1). The following will outline inefficiencies arising from the LINQ-to-objects execution model and, more generally, from querying objects in the memory space of a managed application. Most of these inefficiencies can be assumed to also exist in other applications (e.g., Java's Streams).

- **Virtual function calls** Queries in LINQ are represented by a tree of LINQ's standard query operators. In LINQ-to-objects, each operator is implemented as an iterator that, to produce its result, continuously pulls output objects from its children by calling their `MoveNext()` method and `Current` getter. Queries are evaluated by iterating over the root of the iterator tree, causing the iterators in the tree to step-wise transform the input objects into the query's result. For big data sets, this causes millions of calls to `MoveNext()` and `Current`. Both methods are defined in the `IEnumerator<T>` interface and, therefore, are virtual function

calls. As their target methods are not known at compile time, the compiler cannot inline them, increasing the cost of each function call. On top of the cost of not allowing inlining, virtual function calls cause an indirect branch[1], which impedes instruction pipelining. Iterating over each operator's input using virtual function calls, further, prevents loop based compiler optimizations like loop unrolling or loop pipelining. Iterators cause two virtual function calls per input object per operator. At least another virtual function call per input object is contributed by the function delegate that is passed to each operator as argument to specify the operator's behaviour, e.g., a predicate or transformation function, usually supplied in the query statement as lambda expression.

- **State machine logic** As the iterators that compose the operator tree are state machines, they incur additional per-object instruction overhead to maintain their state.

- **Intermediate result materialization** As all operators in the operator tree operate independently, they have to hand over their result objects to their parent operator. Operators that apply a transformation function on their input (e.g., projection, join or aggregation) have to materialize their intermediate result to be able to transfer result elements to their parent operator. Creating result objects comes at a cost. This cost increases if the creation of a high number of intermediate result objects triggers garbage collection which suspends all application processing for its duration. In many cases, however, these results do not need to be materialized. For instance, consider a sequence of two join operators and an aggregation. Each join operator has to create intermediate result objects to pass to its parent, however, looking at all operators as a single operation, the aggregation could compute its aggregates directly from the triplet of matching input objects of the joins rather than the intermediate result of the last join. This would save having to create the intermediate results of both join operations.

- **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)`. The `OrderBy` sorts its entire input and the `Take(N)` returns the first N objects of the `OrderBy`'s sorted output. LINQ-to-objects first evaluates the `OrderBy` and then returns the first `N` result elements. A better approach would

[1] A branch that, rather than specifying the offset to the next instruction to execute, as is the case for direct branches, specifies where the address is located.

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 another good example to illustrate missed synergies between query operators. In LINQ, aggregation is expressed by a `GroupBy` operator that groups all input elements by a key and either a result selector in the `GroupBy` or a successive `Select` operator that construct the result for each key using one or more aggregate operators such as `Sum` or `Count`. In both cases, each aggregate operator individually iterates over all elements in the group to compute the aggregate. We conducted a simple experiment based on the aggregation in query 1 of the TPC-H benchmark. Our results show that LINQ could process the aggregation 38% faster if it computed all aggregates 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 an additional 12%. Collapsing the grouping and the aggregate computations into a single loop improves the performance by another 10%.

- **No query optimization** There is no query optimization in LINQ-to-objects. Queries that are composed in method syntax are always executed in the order specified by the concatenation of query operators in the method syntax. Queries declared in query syntax are translated into method syntax in the C♯ compiler based on the order that operations are declared in the query statement. There is no optimization stage at compile or runtime that rewrites the query based on heuristics like selection push-down or runtime statistics like histograms to improve query performance. The latter could be collected by the input collection and supplied to the query processor at runtime. Without automatic query optimization, the programmer is required to have a solid understanding of query processing and knowledge about characteristics of the underlying collections to compose efficient queries. Query avalanches [Cheney et al., 2013, Grust et al., 2010] are one of the unfortunate artifacts of this approach when dealing with nested sub-queries.

- **High-level language** LINQ-to-objects is entirely implemented in C♯, whereas database systems are usually written in a lower-level systems language such as C or C++ to give the database engineer greater control over query processing and, in

particular, memory management. In managed languages like C$^\sharp$, all in-memory data is typically stored as objects and managed by the garbage collector. This applies to objects of the source collections of a query, but also to intermediate data structures and results produced during query evaluation. As queries iterate over all objects of their source collections, it is imperative that these objects are stored in memory in close proximity and in the order that they are accessed by the query. Otherwise, the query cannot benefit from CPU prefetching (e.g., cache line prefetching) or may suffer under translation lookaside buffer (TLB) misses. However, this cannot be guaranteed by memory managed by the garbage collector as it is not aware of collections, their content or the order in which contained objects are accessed in queries. Further, the fact that garbage collection may move objects around in the managed heap prevents the query from obtaining direct pointer access to objects, which has negative effects on the performance of `decimal` operations as we will see in Section 3.6. Objects that are allocated during query evaluation for intermediate data structures or results have a limited lifetime, defined by the query and known at compile time. However, garbage collection is not aware of this and, therefore, has to perform garbage collections to reclaim their memory. This causes all application threads to be suspended for the duration of the collection and, hence, slows down query processing. We can improve the query processing performance by manually managing all intermediate data, e.g., in regions [Gay and Aiken, 1998] that use pointer-pushing to allocate data in memory blocks and free all query memory either after query evaluation or at predefined save points during query processing. Storing collection or intermediate data as objects also increases the memory consumption as objects require a 16 bytes per-object storage overhead (on 64 bit systems; e.g., for storing the VTABLE pointer).

The sum of these inefficiencies has a significant impact on the query evaluation performance when using LINQ-to-objects. In this and the following chapters we will try to address each of them to improve the query processing performance of querying objects in the managed heap. The key to achieving this is to leverage query compilation techniques from the database space and refine them to address additional inefficiencies that are unique to managed environments. In Section 3.3, we introduce our architecture to transform LINQ queries into fast imperative query functions that evaluate the query. Our suggested approach is implemented as a library that utilizes LINQ's query provider framework to invoke the query compilation process. Existing applications

can be ported to use our library with minimal changes. In Section 3.4 we look into the generation of pure C♯ code whereas in Section 3.5 we investigate how low-level programming constructs and memory management can contribute to improved the query performance. We then evaluate our approaches in Section 3.6 and see great improvements over LINQ-to-objects.

3.2 Related work

[Murray et al., 2011] first proposed evaluating LINQ queries on in-memory objects by generating imperative C♯ code from the LINQ query statement. Their system utilizes LINQ's query providers to construct a query's expression tree and invoke their code generator. They use an automation-based approach to transform the expression tree via an intermediate representation into an AST representation of a C♯ class with a single method that contains the optimized query. The authors then invoke the C♯ compiler to build a dynamic library from the AST and dynamically load and instantiate it. Their main contribution lies in the automation-based code generator that incorporates two optimizations: iterator fusion and nested loop generation. Iterator fusion is comparable to deforestation in functional languages [Wadler, 1988] and is, in a less formalized way, an integral part of all code generator in database systems for fusing the operations of several query operators into a single loop over the input, e.g., [Krikellas et al., 2010]. Our baseline approach described in Section 3.4 is comparable to this approach, however, produces more efficient code by supporting hash-based joins and by exploiting additional opportunities arising when merging operations of different operators; at the operator and at the source code level.

3.3 Query compilation architecture in C♯

This section outlines the architecture to integrate dynamic code generation into the managed runtime to provide more efficient query processing on data stored in the managed heap as collections of objects. The dynamic code generator transforms the LINQ statements declared in the application's source code into specialized query functions that evaluate the query.

As detailed in Section 2.3.1, LINQ queries that are defined on types that implement `IEnumerable<T>`, e.g., collections, are automatically evaluated using LINQ-to-objects. In Figure 3.1, we outline the code of such a query based on the example of

```
1 void printNamesGreaterAge(List<Person> Persons, int age)
2 {
3   var qry_stmt = Persons
4                .Where(p => p.Age > age)
5                .Select(p => p.Name);
6   foreach (var name in qry_stmt)
7     Console.WriteLine(name);
8 }
```

Figure 3.1: Example of a LINQ query definition and evaluation

Section 2.3.1 wrapped in a function body. Note that, here, the predicate in the `Where` operator (line 4) is parameterized by the `age` variable which is supplied as a function argument. There are several options to automatically transform the query into a specialized query function and execute it instead of the original LINQ query. Our approach uses a custom LINQ query provider to translate the query statement into a dynamically created method that is then executed instead of the LINQ-to-objects operators defined in the query statement. Before giving a detailed description of our approach, we discuss alternative strategies to transform LINQ queries into specialized query functions. This transformation can either be performed statically at compile time or dynamically at runtime.

Note that, in Figure 3.1, the collection type and the entire query statement can be deduced at compile time. The only unknown are the concrete instance of the `List<Person>` collection and the value of the `age` parameter. As the code of the generated query function is independent of both, the query function can be statically generated either by a C♯-to-C♯ compiler before compiling the C♯ source code into IL or directly by the C♯ compiler. The C♯ compiler already translates LINQ queries in query syntax into their method syntax equivalent. This could be extended to generate the query function. Both parameters to `printNamesGreaterAge` are also parameters to the generated query function. However, LINQ also allows query statements to be constructed dynamically which cannot be statically deduced that easily. For instance, if the `Persons` argument in `printNamesGreaterAge` was defined as `IEnumerable<Person>` instead of `List<Person>`, then the method could be called with a collection as `Persons` argument, but also with another query statement that returns `Person` objects. As different calls to the method could pass different query

statements, static code generation is limited to generating a query function for the fraction of the query defined in `printNamesGreaterAge`. Dynamic code generation, on the other hand, sees the entire query (including the query fragment passed through the `Persons` argument) and, hence, can generate query functions for the entire query which enables more efficient query processing. Another disadvantage of static code generation is that the generated code cannot be optimized based on runtime statistics (e.g., by building collection types that incorporate histograms, as is common in database systems) and cannot be recompiled dynamically if changing collection statistics suggest that there is a better way to evaluate the query, i.e., by using a different selection (filter) order. On the other hand, the cost of generating and compiling query functions is non-negligible and static compilation benefits from only having to pay this cost once at compile time, whereas dynamic compilation has to pay it for every query evaluation at runtime. Another advantage compared to our LINQ query provider based approach is that the code of the `foreach` loop that operates on the query result can be pulled into the generated query function and thereby further improve query performance; in particular, if the query has a huge result set and/or if the `foreach` loop contains further refinements to the query result.

An alternative strategy of dynamic code generation is to utilize the just-in-time compiler infrastructure that is already present in managed runtimes. When JIT compiling a method that contains LINQ statements, the compiler can directly generate the query function in its internal representation and replace the original LINQ statement with calls to it. Directly generating the code in the internal representation of the JIT compiler can improve the cost of compiling the query function. However, out of the box, this approach shares the same disadvantages as static compilation as the JIT compiler has no knowledge of the full query statement or about the exact instance of the collection and its runtime statistics when generating the code.

Our approach uses a custom LINQ query provider to dynamically generate, compile and execute the source code for processing a query. Recall from Section 2.3 that queries on collections of objects (which implement `IEnumerable<T>`) are automatically processed using LINQ-to-objects, whereas types that implement `IQueryable<T>` do not use LINQ-to-objects but instead allow query processing through a type-specific LINQ query provider. To bypass the default behaviour of collections and to enable them to use our query provider instead of LINQ-to-objects, we define wrapper classes that implement `IQueryable<T>` around all collection types that we support (e.g., we wrap the `List<T>` collection with `QList<T>`). To utilize our approach, the program-

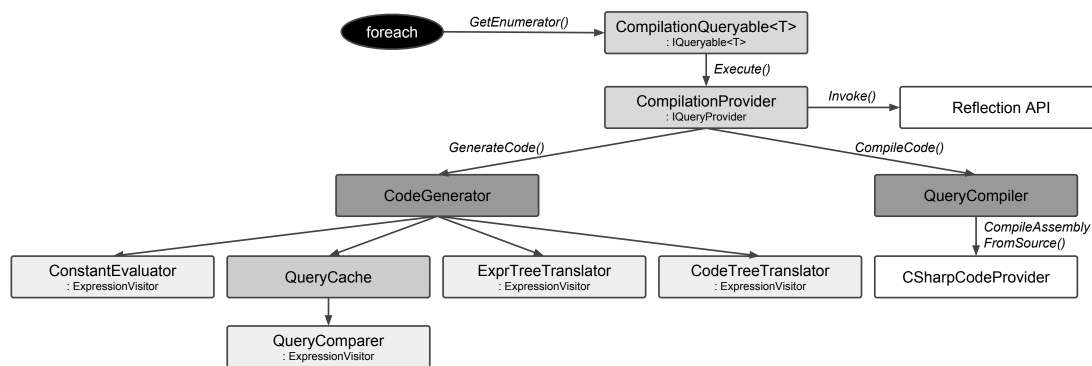


Figure 3.2: Overview of our query provider

mer merely has to replace the original managed collection type with the corresponding wrapper type. This makes our approach very transparent to application developers and facilitates the process of porting existing applications to utilize our approach in order to improve query processing performance.

Recall from Section 2.3.2 that executing the query statement of a LINQ query defined on an `IQueryable<T>` data source, i.e., our collection wrapper, does not evaluate the query, but instead constructs an expression tree representing the operations of the query and returns an `IQueryable<T>` object (`CompilationQueryable<T>` in our case) that contains a reference to the expression tree as well as to the data source's query provider. The expression tree serves as input to our query compiler. Evaluation of the query is deferred to the point where the application consumes a result element. This usually occurs in a `foreach` loop that iterates over the query result or an operation that materializes the query result, e.g., a `ToArray()` or `Count()` operation. However, before evaluating the query, we first have to translate the query's expression tree into specialized source code, compile it using the C♯ compiler, load and finally execute the compiled code to evaluate the query. Our LINQ query provider takes care of all of this.

In Figure 3.2, we outline the processing steps inside the query provider once the evaluation of a query statement is triggered by a `foreach` loop. The `foreach` loop requests an instance of `IEnumerator<T>` from the `CompilationQueryable<T>` object that has been generated from the query statement (held in variable `qry_stmt`) by calling its `GetEnumerator()` method. The queryable objects invokes our query provider `CompilationProvider` to produce the enumerator by calling its `Execute()` method. The `Execute()` method receives the query's expression tree as argument. The query provider then orchestrates the translation of the query tree to a source code representation of the query and the compilation of the generated source code. In the generated

source code, the query is implemented as an iterator method that essentially combines the operations of the query's entire iterator tree into a single iterator over the input. As described in Section 2.3.1, iterator methods are translated into methods that return an `IEnumerable<T>` object by the C♯ compiler. After compiling the generated source code, the query provider uses C♯'s reflection API to locate and invoke the generated method and calls the `GetEnumerator()` method on the `IEnumerable<T>` object returned from the query method. The enumerator is then returned to the `foreach` loop to produce the query's result.

The code generator is invoked by our query provider to translate the expression tree, a tree representation of the standard query operators which is comparable to a query tree in a database system, into the source code of an iterator method. Before generating the code of the iterator method, the query compiler performs several passes over the expression tree to gather information about the query and to apply optimizations by rewriting the expression tree. This part of the query compiler is extensible to allow additional rules to further optimize the generated query code. This is supported by the `ExpressionVisitor` base class which is contained in the base class library and provides a framework to implement custom passes over the expression tree. Our implementation consists of two rewrite rules, the `ConstantEvaluator` and the `QueryCache`. The `ExpressionVisitor` base class allows to add additional rewrite rules to enable more efficient query processing. These rewrite rules could incorporate the following optimizations:

- Changing selection or join orders based on heuristics (e.g., estimated cost to evaluate a selection predicate) and runtime statistics (e.g., histograms) gathered by the underlying collection type.
- The automatic use of predefined data structures that accelerate access to specific data elements such as (B+ tree) indexes or alternative sort orders.
- Automatically decide whether to materialize intermediate or final query results in an in-memory cache and reuse them to answer future invocations of the same queries to improve query performance and response time [Nagel et al., 2013]. In particular for query processing inside applications, queries are likely to repeat as they are statically defined with only a few variable query parameters (e.g., the `age` value in the selection predicate).

The `ConstantEvaluator` traverses the expression tree and identifies all sub-trees that can be evaluated independently of the query's source data (i.e., the underlying

collections). It then evaluates these sub-trees and replaces them with an expression tree node that represents the result. This step also replaces all references to external parameters (e.g., the `age` parameter in our example query) with a node representing the concrete value of that parameter. In addition to improving query performance by only evaluating constant expressions once instead of for every object in the source data, the `ConstantEvaluator` also produces a canonical representation of the query, which is required for the following rewriter rule.

The `QueryCache` contains compiled code of previously evaluated queries together with their expression trees and a hash signature of the expression tree. A second rewriter rule compares the expression tree of the query with the expression trees stored in the `QueryCache` and, if it finds an exact match, uses the already compiled code to evaluate the query instead of producing it again. This saves the time of generating and compiling the query's iterator method, which has been identified by previous work on query compilation in database systems to be a major performance bottleneck [Krikellas et al., 2010]. As we only cache the canonical representation of the optimized query plans, creating a different query plan in an earlier rewriter step because of changed runtime statistics also results in creating a new iterator method instead of using the cached one of the previously used query plan. To efficiently compare the query's expression tree with all cached expression trees we utilize the cached expression tree's hash signatures. All cached expression trees are stored in a hash table based on their hash signature. The first step of the rewriter rule is to compute the hash signature of the query's expression tree in a pass over all of its nodes. The hash signature is dependent on the number and types of the source data and the type and order of the standard LINQ query operators represented by the expression tree. Once the signature is computed, all expression trees in the query cache that hash to the same signature are retrieved from the hash table and each of them is individually compared to the query's expression tree in a top-down pass over both expression trees that checks if the structure and nodes (i.e., node types and values) of both trees are equal. Constant query parameters (e.g., `age` in our example) are ignored to allow the same compiled code to be used for all variations of a query pattern, as long as the parameters do not have an impact on the optimized query plan and, hence, on the generated code. 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 typically are either generated by the application from a limited domain or supplied through user input. Thus, LINQ queries are likely to repeat and reusing

their compiled query code improves the overall query performance and response time of individual queries.

As the cost of finding matching expression trees in the query cache increases with the size of the query cache, the number of compiled queries stored in the cache has to be limited. A limited cache size requires an admission and replacement policy to keep compiled queries in the cache that are likely to benefit the overall performance of the application. Research on automatically caching intermediate and final results during query processing, e.g., [Ivanova et al., 2010, Nagel et al., 2013], has proposed metrics to determine the relative benefit of having a certain result cached. These metrics are based on the cost of producing a result, the saved cost when reusing it from the cache and the expected number of future reuses. The latter is estimated based on uses of the result in the past adapted by an aging factor. In our case, the metrics are simpler as the cost of producing compiled code to evaluate a query can be assumed to be similar for all queries and the benefit of reusing a compiled query from the cache equals the cost of producing it as we compile all LINQ queries before evaluating them. As a result, the benefit metric for the replacement policy is solely based on previous occurrences of the query, aged by lazily multiplying it with a constant factor (e.g., 0.99) for every query received by the query compiler. The admission policy admits all new queries and ensures that they remain cached for a predefined number of query evaluations to prevent cache thrashing of freshly compiled queries.

If the query cache does not contain a compiled version of the query, we have to generate and compile the C♯ code to evaluate the query. Code generation is split into two passes over the expression tree. In a first pass, the `ExprTreeTranslator` translates the expression tree into a *code tree*, an AST representation of the C♯ code of the iterator method that evaluates the query. In a second pass, the `CodeTreeTranslator` translates the code tree into a string that contains the C♯ code of the iterator method. C♯ supports multiple options to dynamically generate and compile code at runtime:

- The `CodeDOM` namespace provides a framework to dynamically build a graph representation of source code elements which can then be compiled using the C♯ compiler.

- The `CSharpCodeProvider` class provides direct access to the C♯ compiler which allows to compile a string containing the source code.

- The `ILGenerator` of the reflection API allows to construct a `DynamicMethod` by directly emitting intermediate language (IL) instructions.

```
1 static IEnumerable<string> Execute( IEnumerable<Person> input_1,
2                                     int param_1)
3 {
4   foreach (Person p in input_1) {
5     if (p.Age > param_1)
6       yield return p.Name;
7   }
8   yield break;
9 }
```

Figure 3.3: Iterator method generated from the LINQ query of Figure 3.1

We chose to generate the iterator method that evaluates the query in a string representation of its C♯ code and then use the `CSharpCodeProvider` class to compile it. Having a string representation of the generated code makes it easier to debug and manually modify it. Directly emitting IL instructions, however, can improve query performance as it bypasses the C♯ compiler and only requires the JIT compiler. We introduce the intermediate step of translating the expression tree into a code tree to reduce complexity when generating the code of the iterator method. Directly emitting C♯ source code from the expression tree can become very cluttered and hard to maintain. By adding the intermediate step, the query operators can be translated into C♯ syntax on a more abstract level without having to deal with concatenating string fragments in the correct order. It also improves the extensibility of the compiler as the compilation back-end (i.e., the second pass) can be exchanged. The IL generator is an example of this.

In Figure 3.3, we illustrate the code of the iterator method that is generated from the LINQ query example of Figure 3.1. Note that the operations of all query operators are combined in a single loop over the query's input (lines 4 to 7). Once the code of the `Execute` iterator method is generated, the query compiler invokes the `CompileAssemblyFromSource` method of the `CSharpCodeProvider` class to compile it. We then create a delegate (C♯ function pointer) to allow easy and fast access to the generated `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, which produces the result of the query by iterating over each result object.

3.4 Compiling to C♯ code

3.4.1 The generated code

We outlined in Section 3.1 that the default implementation to evaluate LINQ queries on collections of objects, LINQ-to-objects, exhibits overheads imposed by its query execution model. LINQ-to-objects composes complex queries by concatenating LINQ's standard query operators and produces the result objects of a query by executing the resulting operator tree. Processing is propagated through the operator tree in a pull-based approach using virtual function calls to transfer processing and data between operators. Operators themselves perform custom processing through function objects that also involve virtual function calls. This execution model poses a high per-object overhead as virtual function calls hinder compiler and processor optimizations. We leverage query compilation to address these inefficiencies. The generated code follows the same principles as introduced in the database space, e.g., by [Krikellas et al., 2010] and [Neumann, 2011]:

- **Specialization** LINQ-to-objects' general-purpose query operators are defined over generic types and use function delegates, usually created by lambda expressions in the query statement, to specify custom behaviour (e.g., predicate or transformation functions). To address the virtual function call overhead caused by the lambda expressions, we replace all generic types with their actual types used in the query and automatically inline the code of all lambda expressions.

- **Operator fusion** To address the virtual function call overhead caused to propagate data through the operator tree, we collapse the entire operator tree into a single, specialized operator that is capable of evaluating the query. To achieve this, we convert the query from a pull-based approach where operators pull their input from child operators to a push-based approach where one or more loop constructs push processing through multiple operators.

Beyond these fundamental changes, the generated code, also addresses some of the other inefficiencies discussed in Section 3.1 such as deeper symbiosis between consecutive operators to save result materializations, unnecessary processing or generally inefficient processing as discussed for aggregation. Apart from this, we mostly do not change the query processing primitives from those used in LINQ-to-object's standard query operators as they already are in line with typical query processing strategies in

database systems. Joins are performed using hash-joins, aggregation is based on hash tables and sorting utilizes quick sort.

We illustrate the conceptual steps to transform the implementations of LINQ's standard query operators into a single, specialized query operator in Figure 3.4 and Figure 3.5 (based on the example of Figure 3.1). As a first conceptual step, each query operator is individually specialized, replacing all generic types with the ones used in the query and inlining the code of all lambda expressions (Figure 3.4b). In a second step, the code of all operators that are involved in the query is combined into a single operator that evaluates the query in one or more tight loops iterating over the query's input or intermediate results (Figure 3.5b). In Figure 3.5a, we present an intermediate step that illustrates the fusion of the `Select` and `Where` operators by combining both in a single iterator method and simulating the control flow that is dictated in the operator tree by virtual function calls to pull output objects from child operators with `goto` statements. By replacing the virtual function calls with `goto` statements, the query evaluation strategy is implicitly transferred from a pull-based to a push-based approach where the `for` loop of the leaf operator pushes objects to its ancestor operators. Note that the operator tree in Figure 3.5a only illustrates the simplified case of two standard query operators. In the more general case, each operator implementation contains an `OP_Pre` and `OP_Post` label and a separate `OP_Next` label for each in- and output. This means that most operations require four labels to simulate the function calls of the original operator tree, while binary operations like joins require five. Take aggregation as an example of a more complex query operator. Aggregation is a blocking operation, i.e., an operation that consumes its entire input before producing its output objects in a second loop, based on an intermediate result produced from the input in the first loop. A hash-based aggregation uses an `Aggr_Input_Next` label to return control to the aggregation when iterating over the input objects to compute the aggregates. It then uses an `Aggr_Output_Next` label in the loop over the computed aggregates to regain control from the parent operator. Note that the steps presented here are only of a conceptual nature, we do not generate the specialized iterator method by refactoring the implementation of LINQ's standard query operators but, instead, directly produce the iterator method from the expression tree representation of the query operators. However, our code generation process that builds the code tree from the expression tree is conceptually based on these observations. In particular, the use of `goto` statements in Figure 3.5a provides a general approach on how to interleave the operations of multiple standard query operators. The query compiler of SQL Server's Hekaton engine

```
1  IEnumerable<Tout> Select<Tin, Tout>( IEnumerable<Tin> input,
2                                      Func<Tin, Tout> selector)
3  {
4    foreach (Tin elem in input)
5      yield return selector(elem);
6    yield break;
7  }
8
9  IEnumerable<Tin> Where<Tin>( IEnumerable<Tin> input,
10                              Func<Tin, bool> predicate)
11 {
12   foreach (Tin elem in input) {
13     if (predicate(elem))
14       yield return elem;
15   }
16   yield break;
17 }
```

(a) Standard LINQ query operators

```
1  IEnumerable<string> Select( IEnumerable<Person> input)
2  {
3    foreach (Person elem in input)
4      yield return elem.Name;
5    yield break;
6  }
7
8  IEnumerable<Person> Where( List<Person> input)
9  {
10   for (int i = 0; i < input.Count; i++) {
11     Person elem = input[i];
12     if (elem.Age > 50)
13       yield return elem;
14   }
15   yield break;
16 }
```

(b) Specialized LINQ query operators

Figure 3.4: Conceptual first step: Specializing LINQ's standard query operators

```
1  IEnumerable<string> Query( List<Person> input)
2  {
3  Select_Pre:
4    goto Where_Pre;
5  Select_Next:
6    yield return elem.Name;
7    goto Where_Next;
8  Select_Post:
9    yield break;
10
11 Where_Pre:
12   Person elem;
13   for (int i = 0; i < input.Count; i++) {
14     elem = input[i];
15     if (elem.Age > 50)
16       goto Select_Next;
17 Where_Next:
18   }
19 Where_Post:
20   goto Select_Post;
21 }
```

(a) Virtual calls replaced with `goto` calls

```
1  IEnumerable<string> Query( List<Person> input)
2  {
3    for (int i = 0; i < input.Count; i++) {
4      Person elem = input[i];
5      if (elem.Age > 50)
6        yield return elem.Name;
7    }
8    yield break;
9  }
```

(b) Specialized query operator

Figure 3.5: Conceptual second step: Fusing LINQ's standard query operators

produces C code that is connected by `goto` statements [Freedman et al., 2014], similar to our conceptual intermediate step in Figure 3.5a.

The specialized iterator method of Figure 3.5b processes the operations of all query operators (here: `Select` and `Where`) in a single loop. This improves query perfor-

mance, by eliminating the overhead of virtual function calls and enables additional compiler and processor optimizations. However, it is not always possible to perform all query processing in a single loop construct. If the query contains blocking operators (e.g., hash join or aggregation) that require all input objects to be consumed before they can produce the first output object, we have to break the current loop and start a new one. In this case, the operation requires an intermediate result (e.g., a hash table) to be materialized in the first loop and then the intermediate result to be consumed in the second loop. Take aggregation as an example. The aggregation iterates over its input in a first loop and, for each input element, it updates the aggregates in a hash table. It then starts a new loop that iterates over the elements in the hash table. As proposed by [Neumann, 2011], we always perform as many query operations as possible in a loop to get the most processing out of every object that is loaded into the CPU cache and touched by the CPU and, hence, maximize query processing performance.

The code generator goes beyond placing the individual operations of successive query operators in the query tree after each other in the generated loop construct. If beneficial for query performance, successive operations are merged. On the operator level, we, for example, merge group by and aggregation operations to save the additional loops over the elements in each group produced by the group by. We also merge successive sort and take (returns the first N objects) operators and process them using a heap of N elements instead of sorting the entire input. Furthermore, we do not create any temporary objects to pass intermediate results between successive operations in the same loop. In LINQ-to-objects, each standard LINQ query operator that does not return input objects (e.g., projections or joins), creates intermediate result objects to pass its result to its parent in the operator pipeline. Instead of following this approach, we delay intermediate result materialization until the end of the loop where we have to materialize the result anyway; either to return an object of the query result to the application or because a blocking operation ends the current loop and, hence, has to materialize the result before starting a new loop on the intermediate result. For instance, take a query that performs several joins and then an aggregation. Following the materialization strategy of LINQ's standard query operators in the loop that performs the probing part of the hash joins and then builds the aggregates in a hash table would lead to creating intermediate objects after each join operation and then again as part of creating the aggregates. Instead, we only create the intermediate objects representing the aggregates and use the input objects of the joins to compute them.

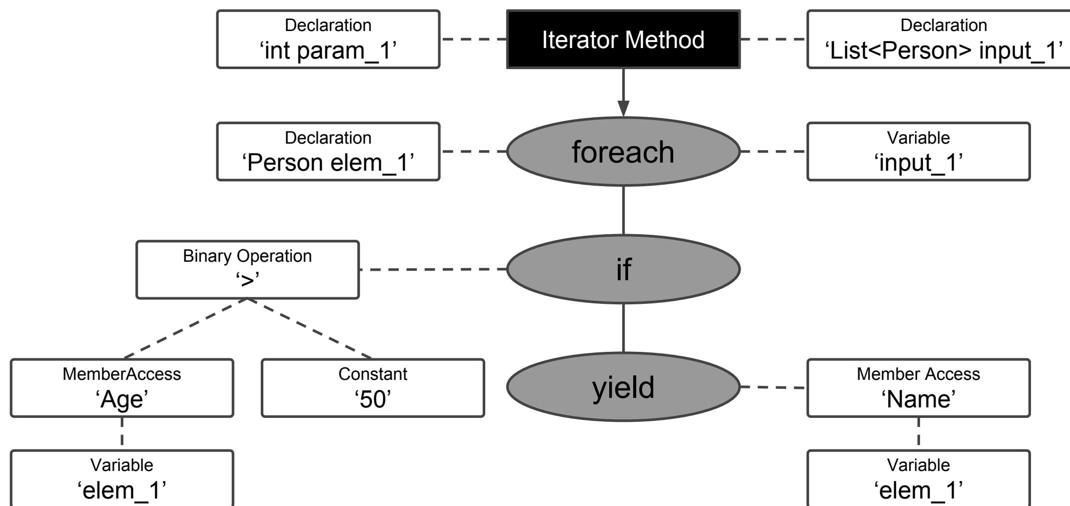


Figure 3.6: Code tree that is generated from the expression tree of Figure 2.6.

We do not cover parallel execution, but because of our database-centric approach, existing parallelization strategies [Dees and Sanders, 2013, Viglas, 2014] are applicable.

3.4.2 The code generation process

The code generator outlined in Section 3.3 utilizes the expression tree which represents the query in terms of LINQ's standard query operators to generate the C♯ code of an iterator method (a specialized query operator) to evaluate the query. In this section, we have a closer look at how the canonical representation of the expression tree is transformed into C♯ source code. A first pass translates the expression tree into a code tree, a tree representation of the C♯ source code of the iterator method. In Figure 3.6, we illustrate the code tree that is generated for the LINQ query of Figure 3.1. Each node represents a C♯ code fragment (`foreach`, `if` and `yield`) and contains additional information required to construct the source code for the fragment. Most of the additional information describing a code fragment can be directly obtained from the lambda expressions that characterize the corresponding operator in the expression tree. More complex operations (e.g., insertions into a hash table) do not have their code included in the code tree, but are instead represented by abstract method calls. The code tree to C♯ string translation then replaces the call with inlined code to perform the operation. The order in which fragments are arranged represents their order in the generated source code. Children are contained in the body of their parents and siblings are placed in order after each other (not present in the example). The order

of arranging code fragments is based of the observations when transforming the implementation of LINQ's standard query operators into a single method using labels and `goto` statements, as illustrated in Figure 3.5a. In this model, each standard query operator is interleaved with the operations of its parent and child operators using four (five for binary operators like joins) labels, `OP_Pre`, `OP_Next_In`, `OP_Next_Out` and `OP_Post`. The points where each operator is interleaved with its parent and children are independent of their operator type and can, therefore, be hard-coded in the code generator for each operator. When traversing the expression tree to create the code fragments for each standard query operator and adding them to the code tree, these fragments have to be added to different parts of the code tree to allow them to be interleaved with those of other operators in the final C♯ source code. We maintain five pointers, derived from the `goto` labels to keep track of where in the expression tree the generated code fragment nodes have to be inserted and, hence, their operations to be interleaved.

The pass over the expression tree also identifies segments that are combined into a single loop construct and the expression tree nodes that border them. These segments either start with the query input or a blocking operation (e.g., aggregation or join) and end with either the root node of the expression tree or another blocking operation. Recall that we only materialize intermediate results that end loop segments. We use the same intermediate result types as specified in the expression tree nodes of the operators that delimit the loop segments. For instance, in the case of aggregation we use the aggregation operator's output type to store the aggregates that make up its intermediate result and in the case of a hash join operator, we use the type of its blocking input (the one used to build the hash table) to store the intermediate result (the hash table). Note that we only create the objects composing the intermediate result if their type differs from the type of the loop input. Using the same type definitions as specified in the expression tree and, therefore, the same naming conventions, allows us to access intermediate result objects as depicted in the expression tree. The nodes that delimit loop segments are either `MethodCallExpression` or `ConstantExpression` nodes, both of which allow type information to be obtained on its in- and outputs. We do not create any other objects representing intermediate results from the expression tree inside a loop construct, but instead apply pending operations on the objects of the loop's final intermediate result. As a result, all objects in a loop construct are either its input or output. The former includes the objects that the loop iterates over, but also objects retrieved when probing hash tables as part of a join operation. To aid the

creation of the code tree for each loop construct, 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. We further track all assignments that the expression tree defines for operations that are part of the loop construct and the name of the variable storing the objects that embody the result of the loop. This allows us to interleave the operations that are performed in a loop construct to avoid the creation of unnecessary intermediate result objects.

Once the code tree is constructed, we traverse it in a final pass to generate the source code string. 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. This improves the compilation time compared to what is common when compiling native C code which usually involves writing the string to a file and then starting an external compilation process to compile it and then loading the compiled binary into main memory.

3.5 Staged query processing

Query compilation in database systems usually generates code in a low-level language (typically C or LLVM). This provides the code generator with more control over how a query is to be processed. In particular, this allows the database system to define the memory layout of the source data (tables) and how to access it in the generated code. It further provides full control of the storage layout and memory management of intermediate query data structures and results. Combined with the use of a powerful ahead-of-time compiler, this allows a highly optimized query binary to be produced in order to execute a query. We believe that utilizing the same techniques on collections of objects can considerably improve the query performance. The basic requirement to achieve such improvements is to be able to access the collections and objects stored therein using C-style pointers. However, this is not possible for managed objects as they are managed by the runtime and may therefore be moved in memory at any time without the application knowing. Accessing a pointer to a memory address inside an object after the object has been moved by a garbage collection would result in incorrect memory accesses. For this reason and to ensure type safety for class types, the .NET framework does not allow direct pointer access to arbitrary collections of objects in the managed heap. Pinning objects in memory to notify the garbage collector to keep their

memory location stable for the time they are pinned would allow for direct pointer access, however is too restrictive for use with arbitrary collections that may contain millions of objects of arbitrary types. However, there are cases where LINQ queries can be processed by native code. Value types in C♯ are not stored in the managed heap but on the stack. Alternatively, C♯ allows unmanaged memory to be allocated using `Marshal.AllocHGlobal` which can then be cast in an `unsafe` environment into a native pointer to a value type. This pointer can then be accessed like an array using indexes. Note that `struct`s in C♯ are value types. If the input data of the query was to be stored in blocks of unmanaged memory containing value types, then LINQ queries could be processed by native code. As the input data is represented as managed objects that are stored in the managed heap, we propose to stage all input data into unmanaged memory before processing it using native code. To do so, we 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 native C code (or `unsafe` C♯; in the following we will use both interchangeably). Because 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.

3.5.1 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 the query 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. To process these parts in C, the data stored in the managed heap first has to be copied into unmanaged memory to allow C direct pointer access when processing the copied data. We look into two options to transfer data to unmanaged memory, materializing all data that is exchanged before processing it in C or incrementally pushing processing from C♯ to C using a buffer. Both the C♯ method that stages the data and returns the query's result objects and the C function that returns the result elements produced in native code are implemented as iterators. However, as query results are assumed to be small, the overhead of pulling result objects from the generated C function is negligible. Alternatively, we could also return result elements from C to C♯ in unmanaged buffers, but to conform with LINQ's deferred execution

strategy, we chose to use iterators.

3.5.1.1 Full materialization

Before performing the heavy lifting of a query in native C code, the data that is to be processed in C is first copied to unmanaged memory. The unmanaged heap is organized as linked lists of blocks of unmanaged memory, i.e., memory allocated outside garbage collection. The generated C code is only invoked once all data has been staged. The staging process is dependent on the query. The query compiler generates the code that performs the staging based on the standard query operators in the expression tree, as was the case when generating the C♯ code to process the query in Figure 3.4. To reduce the volume of data that has to be copied to unmanaged memory blocks, the query compiler decides which parts of query processing are to be performed in the generated C♯ code and which parts in the generated C code. Typically, all selections are pushed into the generated C♯ code as this reduces the number of objects that have to be copied to unmanaged memory. In addition, we perform an implicit projection step in the generated C♯ code to ensure that only data that is actually processed in the generated C code is copied. This is of particular importance because data is copied from objects that may contain references to other objects and copying all data reachable by an object would not be feasible. To copy the object-oriented data to flat unmanaged memory blocks, all object nesting has to be flattened out.

In Figure 3.7, we illustrate the generated C♯ code to stage the data from its input collection and call the generated C (or unsafe C♯) code to perform the heavy lifting of the query. We use C♯'s *platform invoke services* (*PInvoke*) to declare all generated C functions in the class definition (lines 1 to 8) and then use them as if they were local C♯ methods. If the query was to use `unsafe` C♯ instead of C code, these functions would be implemented in the class definition instead of in an external dynamic C library that is accessed using *PInvoke*. As was the case when performing all query processing in C♯, the `Execute` method (line 10) receives the input collections and all query parameters (e.g., the value of a selection predicate) as arguments. It first allocates and initializes a query-specific `Context` structure (line 13) that provides the C code with access to the staged input data stored in unmanaged memory and enumeration state. The generated code then iterates over all input objects (lines 17 to 26), performs the selection on each object to reduce the number of objects that have to be staged (line 18) and finally performs an implicit projection (lines 23 and 24) that only selects object members that are required by the generated C code (`key` and `price`) by only copying

```
1  [DllImport("query0.dll")]
2  public static extern void ProcessInput(IntPtr ctx);
3
4  [DllImport("query0.dll")]
5  public static extern void ProcessIntermediates(IntPtr ctx);
6
7  [DllImport("query0.dll")]
8  public static extern int StreamResult(IntPtr ctx);
9
10 public static unsafe IEnumerable<Tout> Execute( IEnumerable<Order> input_1,
11                                                  DateTime param_1)
12 {
13   Context* ctx = CreateContext();
14   CInput* buffer = AddBuffer(ctx);
15   int count = 0;
16
17   foreach (Order elem_1 in input_1) {
18     if (elem_1.Orderdate >= param_1) {
19       if (count == ctx->elems_per_buffer) {
20         buffer = AddBuffer(ctx);
21         count = 0;
22       }
23       buffer[count].key = elem_1.key;
24       buffer[count].price = elem_1.price;
25       count++;
26   } }
27
28   ProcessInput(ctx);
29   ProcessIntermediates(ctx);
30
31   while (StreamResult(ctx) > 0) {
32     yield return new Tout(ctx->out_elem);
33 } }
```

Figure 3.7: Sample of the generated C♯ code that performs full input staging before invoking C code

their data values to blocks of unmanaged memory. The copying strategy depicted in Figure 3.7 automatically flattens the nested object-oriented data to a flat, row-wise data representation in the unmanaged memory blocks.

The query compiler also generates C♯ code of the `struct` type definition (`CInput`)

for the data stored in memory blocks. This allows the unmanaged blocks to be accessed like arrays of that type and improves readability of the generated code by avoiding manual pointer arithmetic to address the data elements and their fields in the unmanaged memory blocks. Data could also be staged in a columnar fashion by using a separate unmanaged memory block per field (column) and copying each primitive value (e.g., `int` or `decimal`) that is to be staged to the memory block that represents the corresponding field. Once all the data is staged, the generated C code is called to continue processing the query on the staged data. We split the processing performed in C code into several functions. For queries that do not contain binary operators, i.e., no joins, processing is split into three functions: `ProcessInput`, `ProcessIntermediates` and `StreamResult`. For queries that contain binary operators, we generate a `ProcessInput` for each input collection. For every result element returned from C, we create a result object. We assign its members from the unmanaged representation of the result elements, stored in `ctx->out_elem`, and yield the object to the caller. Note that the code in Figure 3.7 only serves for illustration purposes. Instead of using an iterator method, we directly generate the enumerable and enumerator. This enables the use of `unsafe` C♯ code, which is not allowed in iterator methods and to ensure that all unmanaged memory is always cleaned up by placing an additional C call to clean up all unsafe memory into the enumerator's `Dispose` method.

The generated C code for a complex query is outlined in Figure 3.8. Processing is split into three functions. `ProcessInput` processes all staged input blocks until the point where a blocking operation, e.g., an aggregation, ends the loop over the input. The intermediate result of the blocking operation, e.g., the aggregation's hash table, is stored in the query's context structure. The second function, `ProcessIntermediates` starts processing with the intermediate result that has been generated when processing the input in `ProcessInput` and that is stored in the context structure. This function contains all query processing until the final blocking operation of the query. This could contain several loop constructs that each further refine the intermediate result produced by the previous loop. The last intermediate result is again stored in the context to keep it accessible. The `StreamResult` then iterates over the intermediate result, performs all query processing from the final loop construct and returns each result element to the generated C♯ code by assigning it to the `out_elem` field of the context structure. This function is implemented as an iterator. In Figure 3.8, we assume that the last intermediate result is represented as an array. In this case, the `current_pos` field in the context structure maintains the state between successive calls to `StreamResult`. Each

```
1  int ProcessInput(Context* ctx)
2  {
3    MemoryBlock* blk = ctx->input_head;
4    HashTable* ht = ctx->hashtable;
5    while (blk) {
6      for (int i = 0; i < blk->size; i++) {
7        CInput* elem_1 = &(blk->data[i]);
8        /* Process input here, e.g. insert into aggregation hash table */
9      }
10     blk = blk->next;
11 } }
12
13 int ProcessIntermediates(Context* ctx)
14 {
15   HashTable* ht = ctx->hashtable;
16   /* Process intermediate result here, e.g. by sorting it */
17
18   ctx->result = result;
19   ctx->num_result_elems = num_result_elems;
20   ctx->current_pos = 0;
21 }
22
23 int StreamResult(Context* ctx)
24 {
25   if (ctx->current_pos < ctx->num_result_elems) {
26     ctx->out_elem = ctx->results[ctx->current_pos];
27     ctx->current_pos++;
28     return 1;
29   }
30   return 0;
31 }
```

Figure 3.8: Generated C code that processes the query based on the staged input

invocation assigns the next result element to the context's `out_elem` field, increments `current_pos` and returns 1 to indicate that there is a result element accessible through the `out_elem` field or 0 if all result elements have been returned. In Figure 3.8, the actual query processing code is omitted to maintain readability. For instance, the omitted code in `ProcessInput` could update the aggregates of an aggregation by probing and inserting the staged input elements into a hash table. After all input is processed, the

omitted code in `ProcessIntermediates` could iterate over all elements in the previously built aggregation hash table and sort them inside the `result` array. As queries only use data staging if they contain blocking operators and the generated C code is only called for each result object requested by the application, staged query processing maintains support for the deferred execution principle of LINQ: only parts of the query that are consumed by the application have to be evaluated.

So far we assumed that result objects can always be constructed from the output produced by native code, however, this is not always possible. Consider the case where the query result contains references to objects of the query's input. In this case, we cannot copy the reference (i.e., pointer) of the object to unmanaged code. We also cannot copy the object's data to unmanaged memory and then build the result based on a copy of the input object as this changes the semantics expected by the application, especially when the application wishes to modify the input data based on the references contained in the query result. Or consider the case where the query result contains lots of data from the input objects without accessing most of the data in the generated C code. Here, in contrast to the first case, it is possible to copy the data to unmanaged memory and then back to construct the result objects, but it is likely to be too expensive. In such cases, we use the input objects to construct the result. To achieve this, instead of transferring all data to unsafe memory, we only transfer the data relevant for query processing together with a unique identifier that allows the source object to be accessed after query processing to construct the query result. For most input collection types, the unique identifier is already available, e.g., the index of the object in an array or `List<T>` (C♯'s dynamic array) or a unique hash key in a `Dictionary`. If this is not the case, we insert all input objects that are copied to unmanaged memory into a C♯ array or `List<T>` and transfer their indices. To construct a result object of the query in the generated C♯ code, we look up the source object referenced by the unique identifier returned from the generated C code and copy all missing values from it. For instance, 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 processes the query in a similar way, but performs the quicksort in native code on the index and key arrays and then returns result objects in the generated C♯ code in the order defined by the sorted index array.

3.5.1.2 Block-wise materialization

Materializing the entire input of the generated C code suffers from a large memory footprint. Consider aggregating a huge data set to reduce it to a couple of aggregate objects. Only compiling C♯ code as in Section 3.4 does not require much memory as the hash table for the aggregation only contains the resulting couple of objects. However, having to stage all relevant input data before processing the aggregation on the staged data in the generated C code requires a considerably larger memory footprint. To reduce the memory footprint of the staging process, we transfer data from managed memory to unmanaged memory at the granularity of a memory block instead of all data at once. To achieve this, the generated C♯ code stages its input data until a memory block is filled. It then calls the generated C code to process the data in the memory block. Once all elements in the buffer block are consumed by the native code, control is returned to C♯ to prepare the next memory block. The C♯ code can reuse the same memory block for this purpose. As memory blocks are sized to easily fit in the CPU cache, access to the single memory block that is used to transfer data between C♯ and C can be assumed to be very fast. In this case, the additional memory footprint for staging data is bound to the size of a single memory block. However, transferring data in a single memory block does not always contribute to reduce the memory footprint and improve performance. 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 the staged data has to be copied inside unmanaged memory to empty the block and allow the C♯ code to overwrite its content. If this is the case, we rather fall back to the full materialization approach of Section 3.5.1.1 for that input and copy the entire input to unmanaged memory before processing it in C. As a result, the query compiler often uses a combination of full and block-wise materialization. For instance, if the generated C code processes a hash join, then the input collection that is used to build the hash table is fully materialized before calling C to build the hash table, but the input collection used for probing the hash table is block-wise transferred to C.

In Figure 3.9, we show the `Execute` method of the generated C♯ code to process a query similar to that of Figure 3.7. Instead of adding a new memory block to the linked list whenever the current one is full, C is called to process its content (line 11). The code 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 `ProcessInput`.

```
1  public static unsafe IEnumerable<Tout> Execute( IEnumerable<Order> input_1,
2                                                  DateTime param_1)
3  {
4    Context* ctx = CreateContext();
5    CInput* buffer = AddBuffer(ctx);
6    int count = 0;
7
8    foreach (Order elem_1 in input_1) {
9      if (elem_1.Orderdate >= param_1) {
10       if (count == ctx->elems_per_buffer) {
11         ProcessInput(ctx);
12         count = 0;
13       }
14       buffer[count].key = elem_1.key;
15       buffer[count].price = elem_1.price;
16       count++;
17   } }
18   ProcessInput(ctx);
19
20   ProcessIntermediates(ctx);
21
22   while (StreamResult(ctx) > 0) {
23     yield return new Tout(ctx->out_elem);
24 } }
```

Figure 3.9: Generated C♯ code that performs block-wise input staging

```
1  int ProcessInput(Context* ctx)
2  {
3    MemoryBlock* blk = ctx->input_head;
4    HashTable* ht = ctx->hashtable;
5    for (int i = 0; i < blk->size; i++) {
6      CInput* elem_1 = &(blk->data[i]);
7      /* Process input here, e.g. insert into aggregation hash table */
8  } }
```

Figure 3.10: Generated C code that processes the query based on the block-wise staged input

We outline the C code to process the block-wise buffered result in Figure 3.10. Compared to the code generated for full materialization, only the `ProcessInput` func-

tion changes. As it only processes the data of a single block with each invocation, there is no need to iterate over multiple blocks. Everything else remains as before. Note that the intermediate result of the query, e.g., the aggregation hash table, has to be maintained in the context structure to keep it accessible between different invocations to `ProcessInput`.

3.5.2 The code generation process

Generating C and C$^\sharp$ code from LINQ is similar to generating pure C$^\sharp$ code. However, this time the code generator has to split processing between both runtimes based on predefined rules. Recall that the generated C$^\sharp$ code in Section 3.4 is divided into loop segments that were delimited by blocking query operators. When splitting query processing between generated C$^\sharp$ and C code, the inner loops, i.e., all loops in the query that are not on the input or return output elements, are always performed by the generated C functions (`ProcessIntermediates`). The processing of the final loop that returns the query result (`StreamResult`) is also fully processed in the generated C code, but requires additional care to stage data between both runtimes, to create result objects and to possibly look up source objects in order to assign data that has not been staged to the result objects. Processing in the loops over the input, however, has to be split between both runtimes based on heuristics. We assume that the code generator receives a (hand or automatically) optimized operator tree that already has selections pushed down. To reduce the amount of data that has to be transferred to unmanaged memory, these selections are performed in the generated C$^\sharp$ code, together with an implicit projection and flattening step.

We use individual code trees for the generated C$^\sharp$ and C code. While traversing the expression tree, we construct both based on the query operators encountered when traversing the expression tree. In contrast to generating pure C$^\sharp$ code, it is no longer possible for the generated code to directly access all fields of objects from the input collections and (intermediate) results as depicted in the expression tree. The expression tree specifies input and result types in an object-oriented data layout that allows a nested data representation through references to arbitrary data types. The generated C code, on the other hand, relies on a flat, value-type data representation, similar to that of a row-wise relational database, to allow leveraging 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

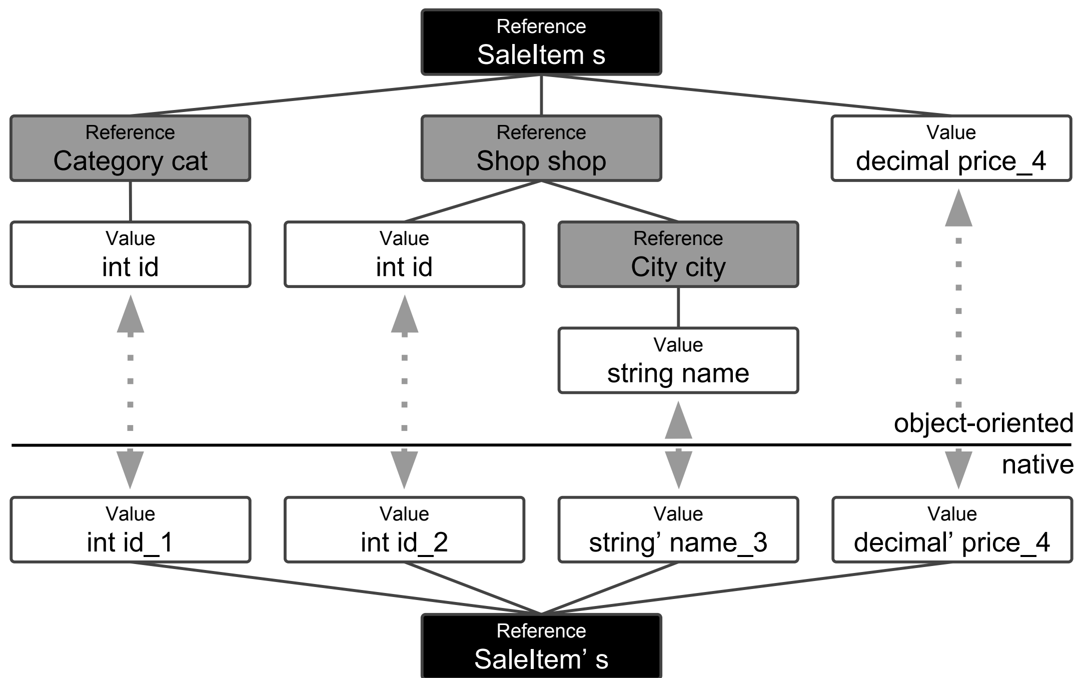


Figure 3.11: Mapping between C♯ and C value types; all value types are stored in-place

between the object-oriented data layouts of the expression tree (i.e., result types of `MethodCallExpression` nodes) and the flat data layouts that the generated C code expects. This allows data accesses depicted in the expression tree to be directly translated to native code that accesses the corresponding data elements in the unmanaged heap. We create `struct` definitions for all data types that are staged between C♯ and C and for all intermediate results produced in C. Doing so improves readability of the generated C code by avoiding unnecessary pointer arithmetic and type casts to access the data. Producing more readable query code facilitates the development of the query compiler as the automatically generated code can be easier read in the debugger. 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.

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 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 unmanaged data layout is represented by one or more trees that each can either be a single value type node or a reference type node which only contains value type children. The unmanaged 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 unmanaged representation (e.g., in Figure 3.11). 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 Section 3.5.1.1), then the reference type to represent these elements maps to the value type that represents the index.

All mappings are created in the same pass over the expression tree that builds the tree representations of the source code. Mappings are created bottom-up. 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., for `Where`, `OrderBy`) or create a new one (e.g., for `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 better 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 Section 3.5.1, 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 the expression tree, we create an empty reference mapping for it. Whenever a `LambdaExpression` in one of the expression's ancestor `MethodCallExpression`

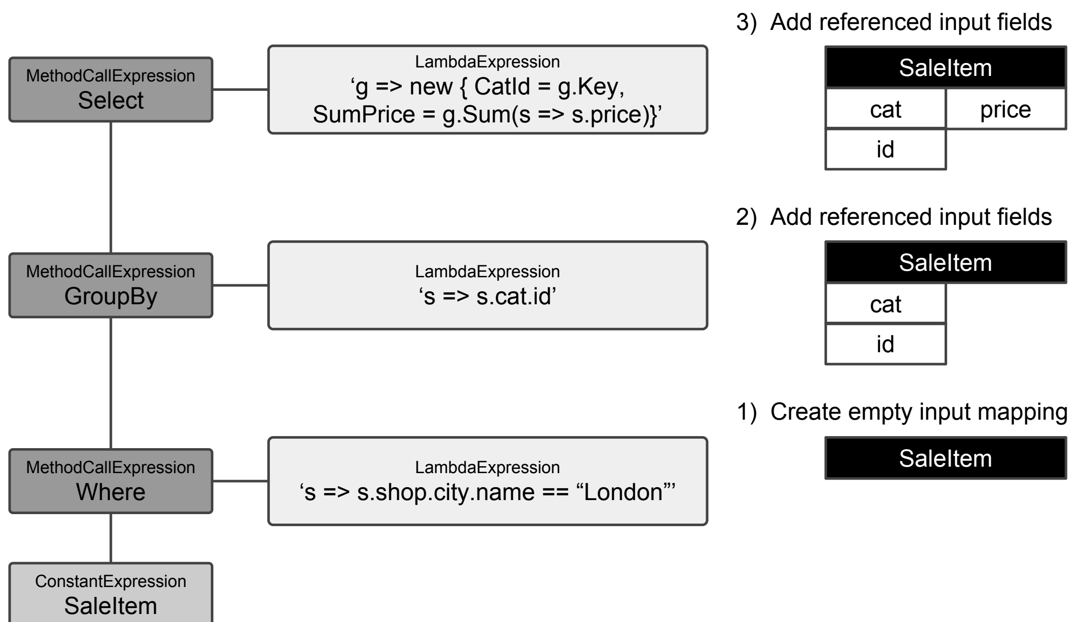


Figure 3.12: Steps of creating an input mapping (bottom-up; selection ignored because it is performed in C♯)

nodes denotes an access to a field of the type stored in the input collection, we check if that field has already been added to the input mapping; if not, we add it. We illustrate this process in Figure 3.12. While navigating up the expression tree, the input mapping is extended with object members that are referenced in the lambda expressions of the expression tree. Note, that *s.shop.city* is not part of the input mapping, because we perform the selection (`Where` in LINQ) in C♯ before copying data to be accessible from native code. We perform the implicit projection step by only copying parts of the input to unmanaged memory that are contained in the mapping.

Once the expression tree traversal reaches the root node and the mapping for the query result is generated, we use reflection to check if the mapping contains any references to input 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 and generate the C♯ code to do so when creating result objects.

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. 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 and group them together.

The staging approach is not used for simple queries that do not contain enough query processing to justify moving managed data to C. There are other occasions when staging cannot be used. This is, for example, the case when the LINQ query statement contains custom logic that goes beyond LINQ's standard query operators. For instance, calling custom C♯ methods during query processing prevents the code generator from porting them to native code and, therefore, it has to fall back to generating pure C♯ code. If the application extends LINQ's standard query operators with custom operators, then neither of our approaches can be applied as the code generator is unaware of them.

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 PInvoke to define a method for calling the generated C function.

3.6 Evaluation

3.6.1 Experimental setup

All experiments in this and the following chapters are performed on an Intel Core i7-2700K system with 16GB of DDR3 RAM, running Windows 8.1 and .NET 4.5.2. The processor has 4 cores at 3.4GHz to 3.8GHz each and supports a total of 8 hardware threads through SMT. It has 8MB of total CPU cache. Unless otherwise specified, all experiments show the average of multiple (typically 7) runs on a warm system. The system was warmed up by running the experiment twice before any performance measurements. All measurements only include the time to execute the compiled query code. We will look into the cost of generating and compiling the code separately after presenting the pure query processing performance.

We evaluate our approaches using the TPC-H benchmark (see Section 2.6) as we believe business intelligence applications represent a common use case of our work.

We adapt the schema to allow us to store the data set in the memory space of the application (e.g., as collections of objects). To achieve this, we translate the schema into an object-oriented representation. Tables become collections. Records become objects and all SQL types are translated into C♯ types that can store the same information; i.e., any character sequence defined in SQL becomes a C♯ string and all other SQL types are translated to their corresponding primitive types in C♯ (e.g., `decimal` or `DateTime`). Note that, since objects store additional information (e.g., VTABLEs) than their payload and most C♯ types are wider than their SQL-equivalent in database systems, the total space required to store the data set is significantly greater than specified by its scale factor (e.g., scale factor one occupies almost 3GB).

As the data set is to be translated into an object-oriented representation, we also translate primary-foreign key relations between records in the database to references connecting related objects. In relational databases, records of different tables are related by one of the tables containing a foreign key column that stores the primary key of the related record in the other table. Queries that intend to access related records together have to perform an explicit join operation between both tables that groups pairs of matching foreign-primary-key pairs together. Joins between foreign and primary keys are very common as records in the fact table are related to records in the dimension tables through foreign-primary key relations. In contrast, the object-oriented model supports these kind of (1:1 or N:1) relationships by defining references from the class type representing the table containing the foreign key column to the class type representing the other table (the direction of the arrows in Figure 2.9). When translating the data set into an object-oriented representation, we omit all foreign keys as references take their place. However, we keep primary keys as they are required for some of the queries.

Translating foreign-primary key relations into references removes the need for queries to perform explicit join operations to access related objects together as they instead merely follow references. As this data layout trivializes some of the join-heavy queries of the TPC-H benchmark, we also test the alternative data representation that uses foreign and primary keys instead of references to relate objects, and, hence has to perform joins to relate objects, as intended in the original (relational) version of the benchmark.

All experiments in this and the following chapters use the scale factor three version of the TPC-H data set, translated into C♯ as described above. We use the first six queries of the benchmark for evaluating our approaches. To evaluate these queries in

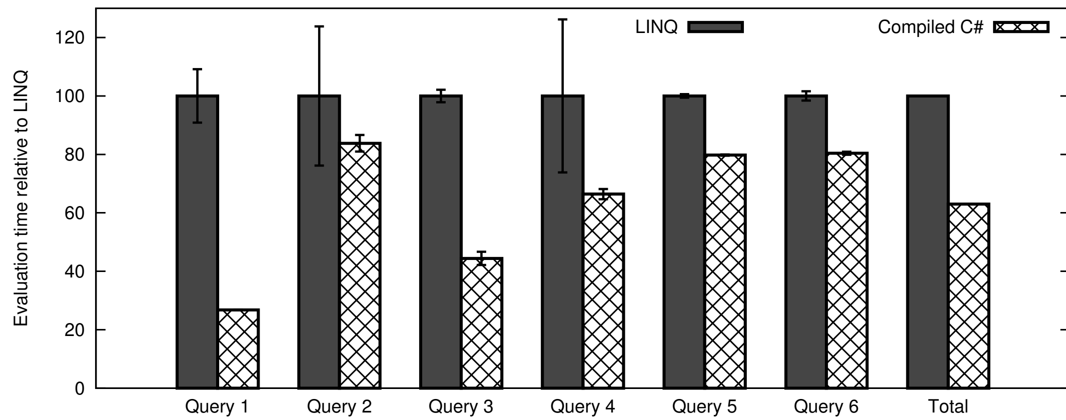


Figure 3.13: Query performance of TPC-H queries 1 to 6 on a reference-based representation of the TPC-H data set

C♯ they are translated from SQL to LINQ statements embedded into the application. Because LINQ-to-objects does not support query optimization as in database systems and instead executes all standard query operators in the order they are specified in the query statement, we declare all LINQ queries in method syntax with query operators concatenated in an optimized execution order. This execution order is based on query plans produced by database systems (e.g., selections pushed down). In the reference-based version, foreign-primary-key joins are replaced by following references (using the *dot* operator, e.g., `lineitem.Order`) and some selection orders are adapted to reflect the cost of following these references. We obtained these orders through manual experiments. The code generator receives these, already optimized, LINQ statements as input.

3.6.2 Compiled C♯

We first look at the query performance when dynamically generating pure C♯ code from LINQ queries. We compare the query evaluation time of the compiled C♯ code with that of the default LINQ-to-objects implementation of .NET for the first six queries of the TPC-H benchmark. In Figure 3.14, we show the result for the reference-based representation of the data set and in Figure 3.13 for the join-based representation. In both cases, compiling the query to specialized C♯ code improves the evaluation time of each of the six queries, other than query 2 for the join-based representation. Note that query 2 contains a correlated sub-query and the reported evaluation times correspond to a hand-optimized LINQ version that avoids executing the sub-query multiple

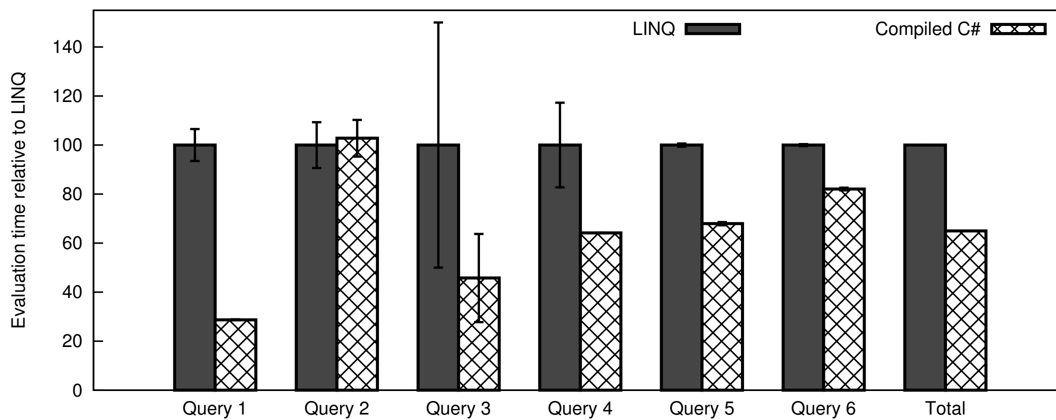


Figure 3.14: Query performance of TPC-H queries 1 to 6 on a join-based representation of the TPC-H data set

times by storing an intermediate result. Directly executing the correlated query in LINQ results in a several orders of magnitude higher evaluation time. The most notable improvements are for query 1 and 3. For some of the queries, the compiled code exhibits a smaller standard deviation. We assume that this is caused by less interruptions due to garbage collections of intermediate query objects in the compiled code.

Query 1 is very aggregation heavy, summarizing a huge input data set into four summarizing result elements that each contain several aggregates. The improvements in query 1 are due to additional shortcomings of LINQ-to-objects when processing multiples aggregates (e.g., `Sum` or `Average`). In this case, LINQ-to-objects computes each aggregate in a separate iteration over the elements of each grouping. These additional iterations over the input data heavily reduce query performance. The generated code processes all aggregates in a single iteration over the input and, further, merges the operations of the `GroupBy` operator with the aggregation to avoid having to create explicit groups beforehand.

Query 3 joins the `lineitem` fact table with the `orders` and `customer` dimensions, performs a simple aggregation, sorts the result and returns the first 10 result elements. The generated C$^\sharp$ code achieves better query performance by merging operations. In particular, merging the sort and the sucessive `Take(10)` operation into a *TopN* operation improves performance by avoiding having to sort the entire result of the aggregation, but instead only maintains a min heap of the greatest 10 elements to process both operations at once.

Overall, the performance difference between the reference-based and the join-based data representation is negligible for queries that do not contain joins (e.g., queries 1

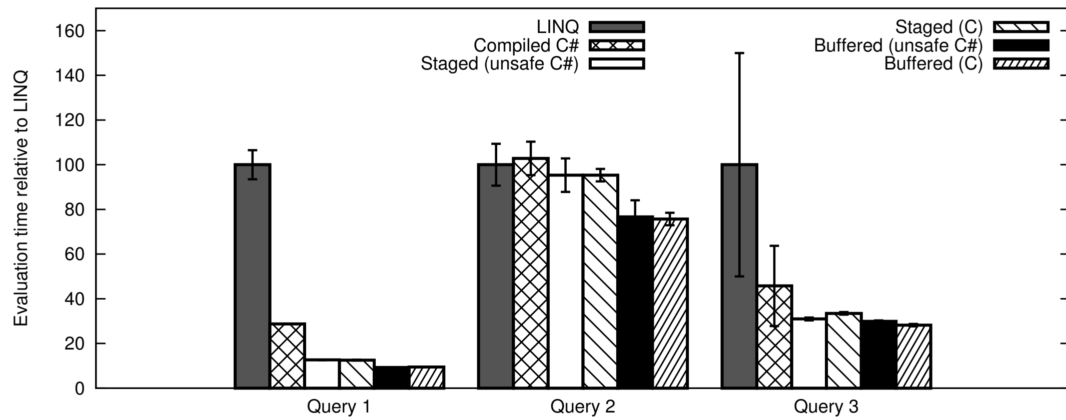


Figure 3.15: Query performance of staged versions of TPC-H queries 1 to 3 on a join-based representation of the TPC-H data set

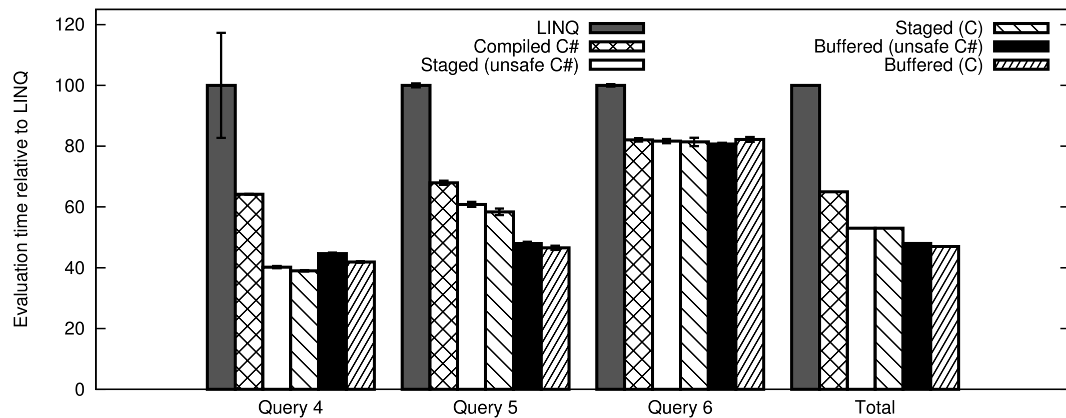


Figure 3.16: Query performance of staged versions of TPC-H queries 4 to 6 on a join-based representation of the TPC-H data set

and 6) as the queries are identical and operate on comparable data sets. For join-heavy queries (e.g., queries 3, 4 and 5), the reference-based data layout is between 40% and 50% faster than the join-based one.

3.6.3 Staged query processing

In Figures 3.15 and 3.16, we compare the query performance of various staging-based approaches to that of compiling to pure C♯ code and evaluating the query with LINQ-to-objects for the join-based data representation. We compare both staging strategies discussed in Section 3.5. The regular approach that fully stages all object-oriented data to unmanaged memory blocks before beginning to process the staged data in native code and the buffered approach that, where possible, continuously stages data between

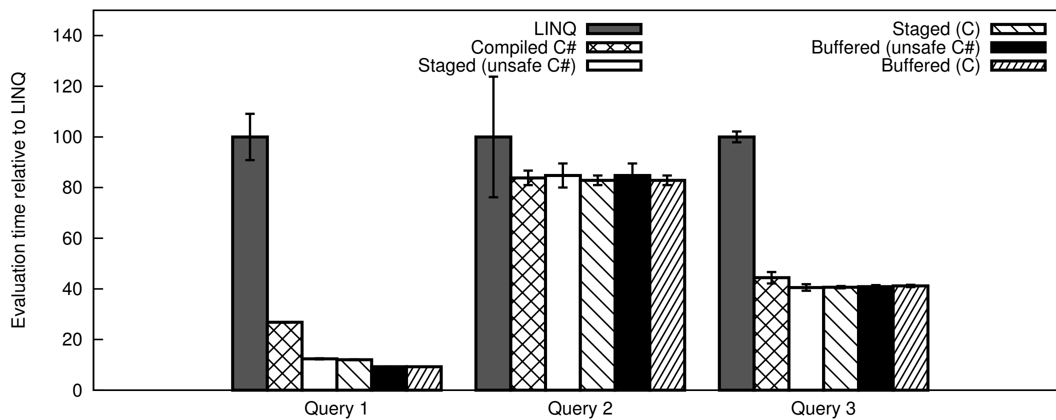


Figure 3.17: Query performance of staged versions of TPC-H queries 1 to 3 on a reference-based representation of the TPC-H data set

managed and native code at the granularity of a single unmanaged memory block. We tested both approaches with generating `unsafe` C♯ and native C code to process the unmanaged buffer blocks.

The potential for improving query performance by staging object-oriented data into unmanaged memory and then processing it in `unsafe` C♯ or native C code depends on how much of the query processing can be offloaded into unsafe code, how expensive the staging process is and how much processing time can be saved by performing parts in unsafe code. We first have a look at the join-based data representation as it allows more query processing (i.e., the joins) to be offloaded into unsafe code and, hence, is expected to produce the greater benefits compared to evaluating the entire query in (safe) C♯. Other than query 6, all queries see improvements in evaluation time compared to that of generating pure C♯ code for all staging strategies. Most queries exhibit better query performance when staging is performed at the granularity of a block. This reduces the memory footprint of the query and improves performance by only using a single unmanaged memory block for staging data that resides in the CPU cache at all times. Query 4 poses the exception to this observation. For most queries, the staged variants exhibit a smaller standard deviation compared to the other cases. This can be explained with the staged variants storing all intermediate query processing data in unmanaged memory and, hence, not requiring garbage collection to reclaim their memory.

Before having a closer look at some of the queries, we first present the results of the reference-based data representation in Figures 3.17 and 3.18. As expected, the benefit of staging diminishes as there is less processing to be performed in native code. Only

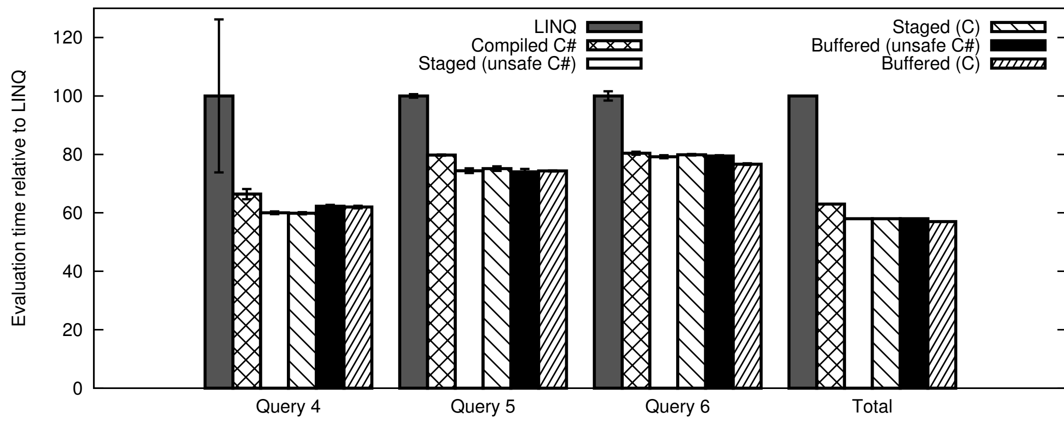
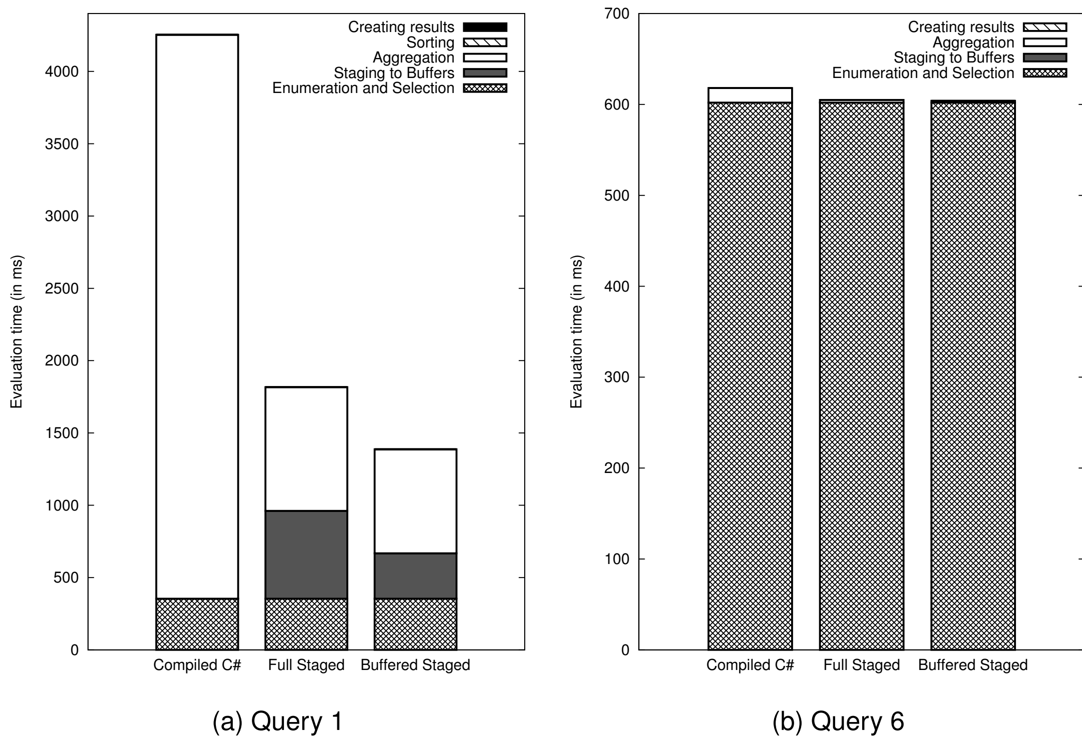


Figure 3.18: Query performance of staged versions of TPC-H queries 4 to 6 on a reference-based representation of the TPC-H data set



(a) Query 1

(b) Query 6

Figure 3.19: Evaluation cost break-down of TPC-H query 1 and 6

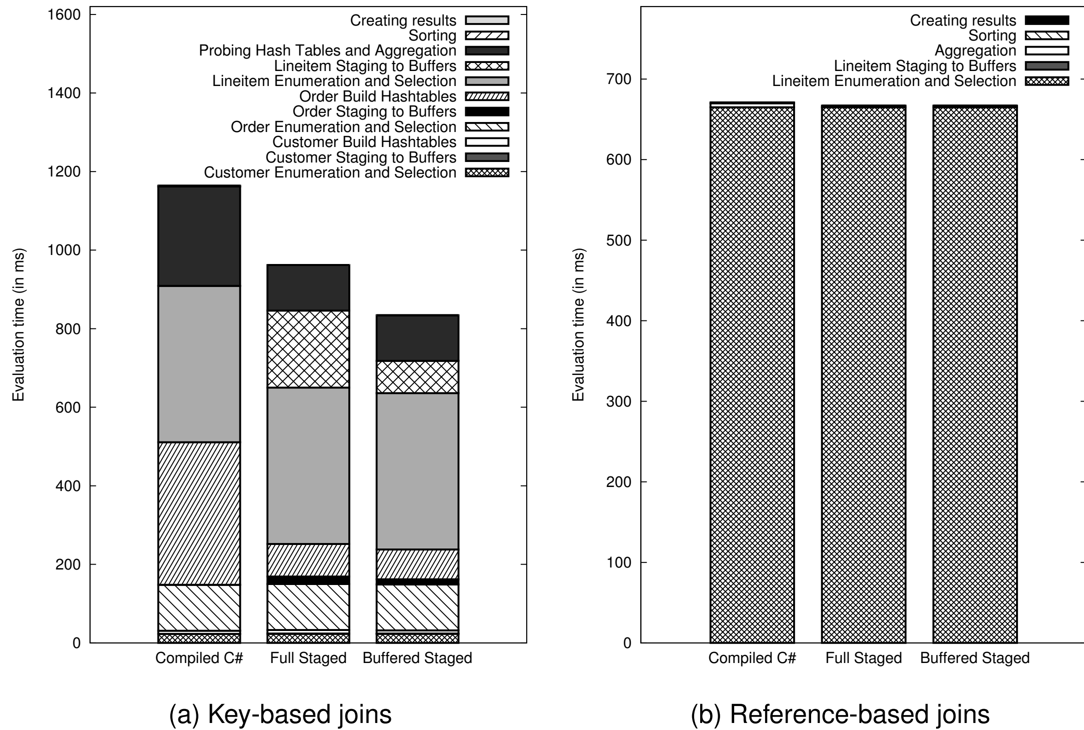
query 1 still shows substantial improvements over pure C♯ code, however, none of the queries show a substantially worse performance either.

We now have a closer look at the cost break-down of some of the queries. Other queries exhibit similar characteristics to the selected queries. Note that the break-down is just an approximation as some of the cost fractions have to be measured in different query evaluations which are manually combined afterwards. First, we have a look at

the two queries which do not contain any joins. In Figure 3.19a, we show the cost break-down of query 1 and in Figure 3.19b that of query 6. Even after removing the inefficiencies that are introduced by LINQ-to-object's execution model in the generated C♯ code, query 1 is still heavily dominated by the cost of computing the `decimal` aggregates. Processing these aggregates in C♯ comes with a disadvantage. As objects may be moved in garbage collection at any time without notice, the methods that compute `decimal` arithmetic cannot directly access the decimal values stored in an object using pointers to the `decimal` value. Instead, they have to be called by value and returned by value. In-place updates are not possible. This poses a huge overhead as `decimal` values are 16 bytes wide and the query computes millions of decimal additions, subtractions and multiplications. By copying these decimals to unmanaged buffers and using unmanaged memory for intermediate results, each only has to be copied once and then can be passed by pointer. In the case of intermediate results (i.e., the aggregates), they can even be updates in-place. In Figure 3.19a, we show how doing so contributes to greatly reduce the aggregation code. It also shows that using a single buffer block reduces the staging cost as all staged data is, at all times, written to and read from a buffer that resides in the CPU cache. As query 1 stages a total of over 1GB of data, this in not the case when staging all data at once before starting to process it in the generated `unsafe` C♯ or native C code.

Query 6 behaves very differently. It is heavily selection dominated. As we perform all selections in C♯ before staging the data and these selections make up for 97% of the total query evaluation time, there is only 3% left that could possibly benefit from being processed in native code. We illustrate this behaviour in Figure 3.19b. Staging reduces the cost of the simple `decimal` aggregation in query 6 for the same reasons as was the case for query 1, but, as the aggregation is only a small fraction of the query processing cost and the total data staged is merely 10MB, the overall result does not show any significant performance improvement.

Query 3 constitutes a representative join-heavy query. In Figure 3.20a, we provide the cost break-down for the query evaluated on a join-based representation of the data set and in Figure 3.20b that of the query evaluated on a reference-based representation. For the join-based version, the two cost fractions that see improvements from compiled C♯ code are the cost of building the orders hash table and the cost of probing that hash table and computing aggregates. The latter part is improved for the same reasons as was the case for query 1. We believe that the cost reduction of building the hash table is mostly due to cheaper allocation of temporary memory from the unmanaged

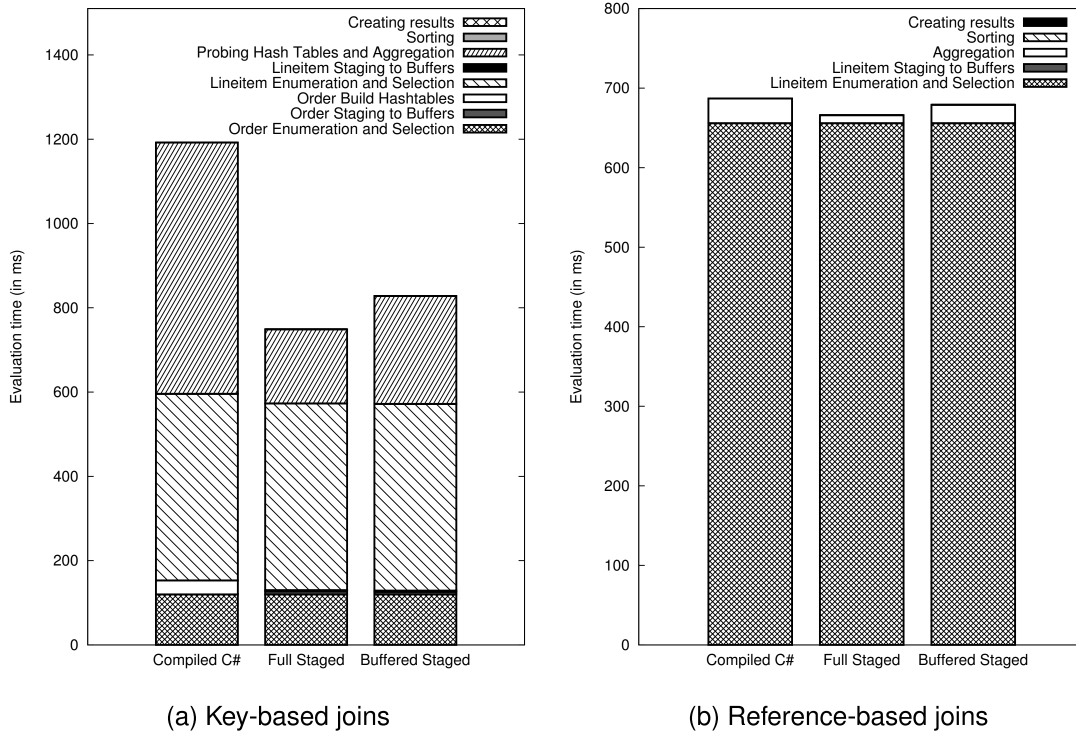


(a) Key-based joins

(b) Reference-based joins

Figure 3.20: Evaluation cost break-down of TPC-H query 3

heap. The hash table on `orders` has to store over two million elements. To keep the comparison fair, we use the same hash table implementation for the generated code as is used by the LINQ-to-objects implementation. This implementation requires around 130MB to store the entire hash table (exclusive of the size of data elements) and some additional space as the hash table dynamically grows. As this data does not fit in the youngest generation(s) of the garbage collector, query evaluation has to endure several short garbage collections which suspend the query processing thread while garbage collection is active and, hence, reduce query performance. Running query 3 multiple times also causes garbage collections in more mature generations. The unmanaged allocator, in contrast, uses a region based implementation, similar to [Gay and Aiken, 1998], that allocates all memory from preallocated memory blocks by merely pushing pointers and frees all memory at once as soon as it is no longer required by the query. Further, as all hash table data is stored in unmanaged memory rather than as objects, the total size of the hash table shrinks to 80MB. Similar to query 1, the cost of staging data is reduced by the buffering approach. Here, a total of 300MB is staged and the reduced staging cost can again be explained by better CPU cache utilization. The reference-based results shown in Figure 3.20b again behave very differently. The cost of accessing the objects through references that may point to random memory



(a) Key-based joins (b) Reference-based joins

Figure 3.21: Evaluation cost break-down of TPC-H query 4

addresses all over the managed heap and, thereby, prevent possible CPU caching and prefetching performance improvements dominates the cost of evaluating the query. The total data staged to unmanaged memory is only 4MB and any performance benefit obtained by the staging process has no impact on the overall query performance.

Finally, we show the same results for query 4 in Figures 3.21a and 3.21b. As before, the aggregation cost in the join-based version (Figure 3.21a) is reduced in the staged versions. However, in contrast to all other tested queries, its cost clearly increases when using the buffering approach. Thorough investigations revealed that the `orders` hash table has a total size of 5MB. By interleaving the enumeration over `lineitem` in the generated C♯ code with the native C code to probe the `orders` hash table and perform the aggregations, as is the case in the buffered approach, the hash table is flushed out of the CPU cache which reduces the query performance because (most) of the hash table has to be re-loaded from main memory with every buffer block processed. The full materialization approach, in contrast, keeps the entire hash table cached at all times and, hence, provides better performance. The referenced-based results, shown in Figure 3.21b, behave similar to what has been the case for query 3 with the cost of accessing objects through references dominating query performance.

So far, we have only looked at the cost of evaluating the query and ignored the

cost of generating query code and compiling it. This is a reasonable assumption as most LINQ queries are hard-coded into the application and only differ in a few query parameters (e.g., the value of a selection predicate). In this case, caching the compiled query after it is produced for the first time and then reusing it on all successive calls to the same query reduces the generation and compilation cost in most cases. However, we report these costs for completeness. Source code generation costs between 30ms and 60ms. C♯ code compilation costs around 75ms and C code compilation around 720ms. To provide a point of reference, the query evaluation time for the six TPC-H queries tested ranges from 105ms (query 2) to 15s (query 1). The cost of finding and reusing a compiled query in the `QueryCache` is negligible. The cost of compiling C♯ code could possibly be reduced by directly emitting IL instructions in the code generator and the cost of compiling C code by generating LLVM code [Neumann, 2011]. As our experiments did not show a significant difference between generating `unsafe` C♯ code and native C code, we will in the remainder of this thesis use both synonymously and only report results for `unsafe` C♯ code. As their generated code is mostly identical, all performance differences come from the different compilers which may change with future releases.

3.7 Summary

This section has shown that query processing in managed runtimes benefits from leveraging query compilation techniques to compile LINQ queries into specialized query functions to evaluate the queries. Query compilation can further benefit from staging the object-oriented data into unmanaged memory blocks to process parts of the query in `unsafe` C♯ or native C code. However, the benefit of all staging approaches is limited by the fraction of the query that can benefit from such an approach. If the cost of accessing the object-oriented data dominates the cost of evaluating a query, then there is not much improvement to be had from staging data.

Chapter 4

Black-box collection

4.1 Introduction

The previous chapter has shown that the performance of query processing in managed applications using LINQ can be improved by leveraging query compilation techniques from the database space. It also demonstrated that there are further performance improvements to be had when processing queries in low-level code and memory management instead of in pure C♯ using automatically managed types. However, the representation of the data set as collections of objects requires the latter and prevents low-level query processing unless an expensive staging phase is added in the generated code. When evaluating low-level query processing using `unsafe` C♯ or native C code in Section 3.6.3, we identified the following main factors improving query performance over using pure C♯ code:

- **Direct pointer access** Providing direct pointer access to operations on a data element's primitive types enables more efficient query processing as opposed to performing the operations on copies of the primitive types. The latter is required for arithmetic functions on primitive types contained in managed objects (e.g., `decimal`) as these objects may be moved by garbage collection and, hence, disallow direct pointer access.
- **Query memory** The lifetime of any intermediate data structure or intermediate results produced during query evaluation is limited by the evaluation time of the query. Furthermore, for every intermediate data structure or result, there is a point during query evaluation where it is known to leave the scope of the query and, hence, can be freed. Using a simplistic memory manager to handle tempo-

rary query memory improves the query evaluation performance for queries that produce large intermediate results. Such a memory manager could allocate intermediate data in memory regions [Gay and Aiken, 1998] using pointer-pushing and free all data stored in a region at once when it is known to have left the scope. Garbage collection based memory management, in contrast, is not aware of lifetime characteristics of memory allocated during query evaluation and, therefore, has to perform garbage collections to reclaim unused memory, which poses additional overhead on query processing. For intermediate data structures or results that contain a huge number of small elements, e.g., the overflow lists of a hash table used in a join operation, the per-object storage overhead of 16 bytes that is required when storing objects in the managed heap poses an additional memory space overhead.

In addition to identifying these opportunities for performance improvements, the experiments in Section 3.6.3 also uncovered limitations of the staging approach that restrict the performance improvement that can be achieved for a query. These limitations are mostly caused by the cost of accessing data stored in collections and applies to both, accessing data in the objects of the primary collection and of secondary collections reached through references. To gain any meaningful performance improvement over the approaches of Chapter 3, these limitations have to be lifted. To achieve this, we have to take garbage collection out of the equation and, instead, introduce a second application heap for storing all data that is used by the application for database-like query processing. This second heap consists of unmanaged memory that enables the performance improvements discussed above and is managed by the collection type that uses it to store contained data elements. Such an approach provides the following, additional performance benefits:

- **Collection-aware memory management** By allowing the collection to manage the memory space of contained data elements, those elements can be arranged in consecutive memory areas in the order in which they are accessed by queries. Doing so improves query performance by making better use of compiler and CPU prefetching (e.g., via cache line fetches) and reduces the likeliness of translation lookaside buffer (TLB) misses. Automatic garbage collections, in contrast, are not aware of the collections in an application, their content or the order in which their objects are accessed in a query and, therefore, cannot optimize the placement of objects in the managed heap based on these charac-

teristics. Objects contained in a managed collection are accessed through an indirection layer (e.g., an array) containing object references. Managing the indirection layer and object placement within the managed heap independently can lead to degraded query performance as collection objects may end up being laid out in the managed heap in a different order than in the indirection layer of the collection. Furthermore, objects that are in close proximity in the indirection layer may end up being far apart in the managed heap. Some implementation characteristics of garbage collectors may worsen this. For example, using the same memory blocks to allocate objects of various sizes as long as they fit in the same size range naturally contributes to object fragmentation. This is also the case when using free lists within managed memory blocks to find slots in which to allocate objects when moving them from a younger to a more mature generation. Finally, the 16 byte per-object overhead on 64 bit systems (e.g., to store the VTABLE pointer) reduces the efficiency of memory prefetching.

- **No impact on GC duration** Storing huge volumes of database-like data in the managed heap increases the maximum duration of garbage collections, independent on whether the collection was triggered by creating objects that represent database-like data or ones that are used by other parts of the application. As objects that are stored in database-inspired collections are assumed to be long-lived, they will most likely end up in the oldest generation and whenever this generation has to be collected, the marking phase first has to perform a depth-first search following all of their references and then the sweep phase has to scan over all their memory blocks to reclaim memory. With data sets at gigabyte scale, this overhead can have a negative effect on application performance. For batch collectors, the extended GC duration results in application threads being suspended for longer amounts of time. For concurrent collectors, it results in the background garbage collection threads consuming CPU and cache resources that could otherwise be utilized by the application for a longer period of time. Storing database-like data in a separate unmanaged heap improves query and application performance by keeping a large fraction of the total application data set outside the managed heap and, hence, excluding it from garbage collection.

- **Long-lived data** Generational garbage collection, as employed by most managed runtimes, relies on the assumption that objects that are created more recently are also more likely to become unreachable. However, this typically does

not apply to objects that are stored in database-inspired collections as they are assumed to be long-lived. Storing them in the managed heap further increases the duration of garbage collections as most of these objects survive all collections in younger generations and, therefore, are gradually copied from the youngest to the oldest generation which further increases the garbage collection overhead.

- **Decoupled storage layout** As data elements are managed at the granularity of a collection that contains huge numbers of elements of a single type which are predominantly accessed through LINQ queries, we can decouple the internal storage layout from the type definition. In particular, introducing columnar storage layouts [Copeland and Khoshafian, 1985] instead of row-wise layouts, as is the case for managed objects, has been shown in the database space to improve query performance [Manegold et al., 2000].

In the following chapters, we will introduce two collection types that make use of an additional application heap for storing unmanaged data. In both cases, the collection type itself is automatically managing the memory space of all data that is stored in the collection. Furthermore, in both cases, language-integrated query is assumed to be the dominant means of accessing collection data. In this chapter, we introduce *black-box collections* which store data in a database-inspired in-memory data store inside an unmanaged heap that does not allow the application to directly access the elements in the underlying data store (hence, *black-box*). In Chapter 6, we introduce *self-managed collections*, a collection type that stores manually-managed objects in the unmanaged heap. Self-managed collections rely on a safe manual memory manager that will be presented in Chapter 5.

Black-box collections are designed to provide a query processing performance that is comparable to that of a modern relational database system. To achieve this, we no longer use the runtime's automatic memory management system and, hence, no longer store data as managed objects. Instead, black-box collections employ a database-inspired in-memory data store that is hidden from the application. Only the collection type and automatically generated query code have direct access to data elements inside the data store. From the perspective of the application, the collection stores objects of a managed type and the application code interacts with the collection through objects. This illusion is achieved using existing object-relational mapping techniques.

Data elements inside black-box collections are assumed to be predominantly accessed through LINQ queries. Furthermore, these queries are assumed to rarely return

elements of the collection's data type but, instead, return types that represent refinements of the base data. We believe that this is a common use case for many applications as the results of LINQ queries are often used to generate the application output (e.g., as tables, charts or graphs) and, hence, tend to be rather small im order to not overwhelm the user. For instance, most queries in analytics workloads make heavy use of aggregation to condense their input data into a few summarizing elements. Based on these assumptions, our design of black-box collections focuses on improving the query evaluation performance, if necessary, even at the cost of the performance of returning or modifying collection data. To provide fast query processing, we integrate the data store of a relational database system into the unmanaged heap and allow our code generator to translate LINQ queries over black-box collections into `unsafe` C♯ or native C code that directly processes the query on the underlying data store rather than on the object-oriented representation that the application uses to interact with the collection. This implies that the generated query code directly operates on the memory blocks of the data store rather than on an indirection level as is the case for regular collections. By storing collection data in consecutive memory addresses and allowing queries to iterate over collection elements in the order they are stored in memory improves the effectiveness of compiler and CPU prefetching and, hence, the query performance. As black-box collections use the same data layout and, by using similar query compilation techniques as [Krikellas et al., 2010, Neumann, 2011], also the same query processing techniques as in database systems, the query processing performance can be assumed to be comparable to that of relational database systems. Compared to the approaches of Chapter 3, we no longer store data as collections of objects and, hence, provide better query processing performance by addressing the issues listed above.

Black-box collections essentially integrate the in-memory storage layer of a database system into the managed runtime. Compiling LINQ queries into low-level source code that directly operates on the data store allows a deep integration of the data store and the managed runtime where the boundaries of both are continuous. Utilizing the storage of a database system allows to easily incorporate more advanced database techniques such as the collection of runtime statistics (e.g., histograms), clustering the data store or defining indexes on the data. With LINQ queries on black-box collections being parsed into an expression tree representation that resembles query trees in database systems and allows to rewrite it in multiple passes before generating the source code to process the query as described in Section 3.3, we can employ query optimization like in database systems to improve the quality of the generated query code based on the

collected runtime statistics and the availability of indexes or orders in the underlying collection.

Instead of storing managed objects, black-box collections store data elements in an in-memory data store in a manner that is derived from how relational database systems store the data contained in a database table. As a result, data elements stored in black-box collections do not support references to directly connect related objects and, hence, do not allow nested data representations. Instead, relationships are modeled as in relational database systems where an object stores the primary key of a related object as foreign key. Explicit primary-foreign-key joins are used to access related data elements. Employing this strategy facilitates memory management as there are no references between elements in the data store and, hence, removing or relocating a data element does not leave any dangling references. However, performing explicit join operations comes at a cost and may cripple query processing for join-heavy queries.

Application code that uses black-box collections cannot directly interact with the data elements stored in the collection. Instead, managed objects serve as proxy between application and collection. The collection provides the application with an illusion of containing objects by allowing the application to interact with it as if it was a managed collection that contained objects of a predefined type. This is enabled using existing object-relational mapping techniques. From the perspective of the application, black-box collections are defined on a managed type and contain objects of that type. The collection's insertion and removal methods take an object reference of the collection type to insert (or remove) the corresponding data elements into (from) the collection's data store. Black-box collections implement the `IQueryable<T>` interface, where `T` is the managed type that the collection is defined on, to allow LINQ queries over black-box collections to be formulated using the collection's managed type. However, the generated code to evaluate the query operates directly on the underlying data store and employs the object-relational mapping to create managed objects when returning data elements stored in the collection. Mapping data elements between both data representations causes overhead when exchanging data between the data store and the application. Based on our assumptions about typical use cases for black-box collections, the performance improvements gained by more efficient query processing should compensate for these overheads. As black-box collections have a different internal representation to the object-oriented one used in the application, some of the issues that applications face when integrating external relational database systems, as discussed in Chapter 1, are also present for black-box collections.

4.2 The basic collection type

Black-box collections manage the memory space of contained data in a similar manner as data stored in tables is managed in relational database systems. As such, they can be seen as the storage layer of an in-memory database system integrated into the managed runtime. Despite storing relational data, the application code interacts with black-box collections as if they stored the managed type that is defined by the collection's generic type argument. We assume existing object-relational mapping techniques, in particular LINQ-to-SQL (see Section 2.4), to be used with black-box collections to achieve this. However, as our main focus is to show the performance benefits of processing LINQ queries using a database-like data store and compiling to low-level code, we will only have a brief look at object-relational mappings. The following code illustrates the basic use of black box collections:

```
CollectionContextImpl mydata = new CollectionContextImpl();
Person adam = new Person("Adam", 27);
mydata.Persons.Add(adam);
/* ... */
mydata.Persons.Remove(adam);
```

Note the similarities to using managed collections like `List<T>`. The only difference is that all collection instances are declared in the `CollectionContext` (`DataContext` in LINQ-to-SQL) implementation. This is the case because the object-relational mapping framework requires prior knowledge about the collection instances that foreign keys refer to. LINQ queries on black-box collections also exhibit the same syntax as queries on conventional collection types. However, differences occur when defining the classes used with black-box collections as they have to be annotated with additional information like key constraints to allow the object-relational mapping system to automatically map between the object-oriented and the relational representation, as illustrated in Figure 2.7. Additionally, updates to the underlying data store are handled differently as an explicit `SubmitChanges` call is required to write pending changes performed on the object-oriented representation of the data back to the collection's data store. The lifetime of all data elements stored in black-box collections is from their insertion into the collection to their removal. The lifetime of their object-oriented equivalents is as long as they are reachable through references, but modifications can only be written back to the collection's data store while the `CollectionContext` instance that retrieved the data exists and the data elements are not removed from the collection's data store.

We only assume common object-relational mapping techniques and no deeper integration between a collection's data store and the managed application code to allow the collection to manage the memory of the data store as in a relational database system without imposing any external restrictions (e.g., by pointer to elements in the data store). However, in what follows, we will outline the implementation and semantics of an alternative version of a LINQ-to-objects `DataContext` to allow a deeper integration of the relational data store and the object-oriented application. This alternative version is enabled by the relational data store residing in the memory space of the application rather than in an external database system. Our implementation also uses an identity cache to store references of all objects returned from the data store; however, instead of each element in the cache containing the pair `<primary key(s), reference>`, we instead store the triplet `<primary key(s), weak reference, pointer>`. Note that the reference to the returned object is turned into a weak reference and a pointer field is added. The pointer field contains a pointer to the data element in the data store that the returned object was created from. To allow the identity cache to store pointers to elements in the data store, these elements may not be moved inside the data store (e.g., to compact non-full memory blocks) while being in the identity cache. We achieve this by *pinning* the elements in the data store. As the identity cache is implemented as a hash table on the primary key(s), the addition of the pointer provides fast access from an object to its underlying element in the data store. We use this to directly write back all changes performed on objects returned from the data store when they occur instead of requiring the programmer to issue `SubmitChanges` calls to update the data store. This can be achieved by using automatically generated setter methods, similar to how LINQ-to-object's `DataContext` tracks changes to returned objects. As all changes to returned objects are automatically written back as they occur, we no longer have to keep these objects alive for the lifetime of the `DataContext`. This is where weak references come in. Weak references store a reference to a managed object, but in contrast to regular (i.e., strong) references, they do not prevent garbage collection from reclaiming their memory space. If the referenced object has been collected, the weak reference becomes `null`. By only storing weak references in the identity cache, we no longer prolong the lifetime of returned objects. Once a returned object is no longer reachable by application code, it is collected. Automatically generated `finalize` methods are used to remove objects from the identity cache and to unpin data elements in the data store when the corresponding object is collected. Alternatively, this could also be performed periodically by scanning over all elements in the identity cache and remov-

ing the ones where the weak reference is null. As this approach limits the number of returned objects that are kept in memory, we can allow an instance of the adapted DataContext to be used for the entire lifetime of the application rather than requiring to create a new one for each individual unit of work. By making this instance a singleton, we prevent the existence of any other instances and, hence, ensure that all application threads operate on the same object instances and, therefore, see each other's modifications.

Black-box collections are implemented using unsafe C♯ code to manage their data store. Each collection contains a linked list of unmanaged memory blocks that contain collection elements. As was the case for the unmanaged memory blocks used to stage data between managed C♯ and low-level code in Section 3.5, we use the AllocHGlobal method from C♯'s Marshal class to allocate blocks of unmanaged memory. We use a C♯-to-C♯ compiler to automatically add struct definitions for all managed classes that are used with black-box collections. Furthermore, we generate conversion functions for each of these managed types that copy the data stored in a managed object to its unmanaged representation or vise versa. We also add a static getter method to the managed type that returns the size of a data element in the unmanaged representation. The collection's Add method uses these methods to access the unmanaged data store and to write data from the supplied managed object to the data store's memory representation. Furthermore, the query utilizes them to create a managed object from the unmanaged representation that has been returned from the data store, in case the query returns elements of an input collection and the corresponding object is not yet stored in the identity cache.

4.2.1 The generated code

Black-box collections provide fast query processing performance through the symbiosis of dynamic query compilation and an unmanaged in-memory data store that is organized as in a database system and that allows the generated code direct pointer-based access to the memory blocks of the data store. To allow the generated code to access elements in the data store using C-style pointers, we compile LINQ queries into either unsafe C♯ or native C code. In both cases, we generate a specialized query function that allows the application to iterate over the query result using an IEnumerator<T>, as was the case for the approaches of Chapter 3. The query function constructs managed C♯ objects from the unmanaged query result produced by the unsafe C♯ or native

```
1  [DllImport("query0.dll")]
2  public static extern int EvaluateQuery(IntPtr ctx);
3
4  [DllImport("query0.dll")]
5  public static extern int CleanUp(IntPtr ctx);
6
7  public static IEnumerable<string> Execute( BlackBoxCollection<Person> input,
8                                              int param_1)
9  {
10   IntPtr ctx = CreateContext(input->data, param_1);
11
12   while (EvaluateQuery(ctx) > 0)
13     yield return CreateResultObject(ctx);
14
15   CleanUp(ctx);
16   yield break;
17 }
```

Figure 4.1: Generated C♯ code wrapping calls to the query function written in native C code

C code. If the query result consists of elements from the input collection's data store, then the identity cache (as described in Section 2.4 for LINQ-to-SQL) is probed and, only if the object is not cached, it is created and its values assigned from the unmanaged result. If the query result does not consist of elements from the input collection, the result objects are created as specified in the query's expression tree based on the values from the unmanaged result. The latter is assumed to be the most common use case.

In Figure 4.1, we outline the code of the generated C♯ wrapper class that serves as glue between the application and the generated C code to evaluate the query. As was the case in Section 3.5, the generated C code is compiled into a dynamic C library and the generated C♯ code declares the generated C functions in the class definition using PInvoke to allow the C functions to be used as if they were C♯ methods. If the query was to use `unsafe` C♯ rather than C code, there would be no need for a second iterator as all query processing could be implemented in a single iterator. The `Execute` method (line 7) is again implemented as an iterator method to allow the application to iterate over the query's result objects. However, here, it does not perform any query processing itself, instead merely acts as glue between the application and the generated

```
1  int EvaluateQuery(Context* ctx)
2  {
3    MemoryBlock* blk = ctx->current_blk;
4    while (blk) {
5      for (int i = ctx->current_pos; i < blk->size; i++) {
6        Person* elem_1 = &(blk->data[i]);
7        if (elem_1->Age > ctx->param_1) {
8          ctx->out_elem = elem_1->Name;
9          ctx->current_pos = i + 1;
10         return 1;
11     } }
12     blk = blk->next;
13     ctx->current_blk = blk;
14   }
15   return 0;
16 }
```

Figure 4.2: Query function written in native C code

low-level code that processes the query. The `Execute` method receives the black-box collections it operates on as well as all query parameters. It first allocates and initializes a query-specific *context* structure (line 10) that provides the C code with access to the unmanaged data store of each input collection, the parameters of the query and the enumeration state. The `EvaluateQuery` C function is also implemented as an iterator that, for each unmanaged result element, sets the `out_elem` field of the context structure to point to that result element and returns 1. The `Execute` method continuously calls the `EvaluateQuery` C function (line 12) to produce the next result element until it returns 0 to indicate that there are no further result elements. For each unmanaged result element returned by `EvaluateQuery`, `CreateResultObject` (line 13) constructs the corresponding managed result object and returns it to the caller. This evaluation strategy maintains support for the deferred execution principle of LINQ: only parts of the query that are consumed by the application have to be evaluated. Once the entire result has been produced, `CleanUp` (line 15) is called to give the generated C code a chance to free all the memory it used to produce the unmanaged result.

The generated C code follows the same principles and exhibits the same optimizations as discussed in Chapter 3. However, instead of iterating over collections of ob-

jects (as in Section 3.4) or over staged buffers (as in Section 3.5), it iterates over all unmanaged blocks in the input black-box collection and process all data elements stored in each block. In Figure 4.2, we illustrate the generated C code for the example LINQ query of Figure 3.1. The query function iterates over all data elements (lines 4 and 5), checks if the age is greater than the supplied parameter (line 7) and if this is the case, assigns the person's name to the `out_elem` field of the context structure (line 8) before returning control to the generated C♯ code to create the result object and return it to the caller. The `current_blk` and `current_pos` fields in the context structure are used to split the iteration over the input data into several function calls. Before returning, both are set to the next element of the input to continue processing from that element in the next `EvaluateQuery` call. The context structure also holds all local variables that need to be kept between different calls to `EvaluateQuery` (e.g., pointers to hash tables).

4.2.2 The code generation process

The code generation and compilation process for LINQ queries on black-box collections is very similar to that of generating C♯ and C (or `unsafe` C♯) code when staging data in C♯ and processing the staged data in the generated C code (as discussed in Section 3.4). The main difference is that all query operations are mapped to the generated C code and the generated C♯ code merely acts as wrapper between both runtimes. Furthermore, result objects are created differently as the identity cache has to be consulted if the query result consists of elements from the input collections.

The generated C code contains the `struct` definitions of all unmanaged types that are used to represent the elements of the input collections of the query. They are comparable to the `struct` definitions that are included in the generated C code to access the unmanaged data stored in buffers in the staging approach of Section 3.5. However, as our C♯-to-C♯ compiler already automatically creates the `struct` definition for the black-box collection implementation to manage the data store, we reuse it when generating query code. To achieve this, the C♯-to-C♯ compiler adds a static getter method to the class definition of the corresponding managed type, the generic type parameter of the collection, that returns a string representation of the `struct` definition's C code.

```
1  int EvaluateQuery(Context* ctx)
2  {
3    MemoryBlock* age_blk = ctx->current_blks[0];
4    MemoryBlock* name_blk = ctx->current_blks[1];
5    while (age_blk) {
6      int* Age = (int*)(&age_blk->data);
7      char** Name = (char**)(&name_blk->data);
8      for (int i = ctx->current_pos; i < age_blk->size; i++) {
9        if (Age[i] > ctx->param_1) {
10         ctx->out_elem = Name[i];
11         ctx->current_pos = i + 1;
12         return 1;
13     } }
14     age_blk = age_blk->next;
15     name_blk = name_blk->next;
16     ctx->current_blks[0] = age_blk;
17     ctx->current_blks[1] = age_blk;
18   }
19   return 0;
20 }
```

Figure 4.3: Query function on top of columnar data store

4.3 Columnar storage

Black-box collections group data elements into unmanaged memory blocks of the same type and collection. The data elements are organized in a similar way as in a relational database system which also implies that they are inaccessible by application code and are only accessed by the collection or through code generated from LINQ queries. As such, black-box collections allow us to decouple the layout of the managed type that the application uses to interact with the collection and the memory layout that the collection uses to store the data elements in the collection's data store. So far, we used a row-wise data layout in the data store as this closely resembles the internal layout of objects where all fields are stored in consecutive memory addresses. However, as outlined in Section 2.1.3, for many workloads, a columnar data layout can lead to superior query processing performance.

It is fairly straightforward to create a columnar version of black-box collections.

Instead of each collection's data store containing a single linked list of memory blocks that each contain data elements defined by the `struct` type derived from the managed type of the collection's generic type parameter, the data store contains multiple linked lists, each representing a columns and containing the primitive type values of that column. In our implementation, we use different block sizes for the blocks of each column to insure that the memory blocks of all column store the same number of elements. Doing so reduces the loop overhead when iterating over the data elements as the values of all columns can be accessed with the same index variable and the iteration reaches the last element of the block at the same time for all columns. The code generator has to be aware of columnar black-box collections in order to produce code that accesses the data store in a columnar fashion. In Figure 4.3, we outline the C code generated to process the query of Figure 3.1 on top of a columnar black-box collection representing the `Person` type. The query context contains an array (`ctx->current_blks`) that stores pointers to the current memory blocks of all input columns that are accessed by the query. The generate query code iterates over these blocks, casts the data in each block to an array of the correct primitive type and accesses the values of each column using its array index.

4.4 Related work

4.4.1 Building database systems in high-level languages

Black-box collections aspire to integrate some of the functionality of database systems into the memory space of a programming language to enable the application to perform faster query processing without the need of a full-blown external database system. At the time of publishing this approach in [Nagel et al., 2014], [Klonatos et al., 2014] proposed using comparable techniques to write the entire database system in a high-level programming language (Scala) and utilize query compilation techniques to lower the entire database system in several steps to the level of native C code that is then compiled using the existing compiler infrastructure. In this section, we describe our take on this idea and how we see it fits with our approaches by outlining an example DBMS that, like our approaches, is written in C♯ and utilizes existing .NET components as much as possible.

Implementing database systems in a high-level language improves the productivity of the database engineer and results in a code base that is easier to maintain and extend.

For example, the query execution engine of the database system can be implemented by using the same high-level language constructs as are used for LINQ-to-objects' implementation of the standard query operators. This facilitates the work of the database engineer as she no longer has to deal with the low-level aspects of the query like input type interpretation or pointer arithmetic. In fact, the database system could even start off by using the actual LINQ-to-objects implementation. Since the input of a database system are SQL queries represented as strings, the system has to parse them into a tree representation of the query operators. As the operators are the LINQ-to-objects operators, the expression tree seems a reasonable representation of the query. This further increases productivity compared to low-level code as the existing `ExpressionVisitor` base class can be utilized to define rewrite rules to perform query optimization.

The database system discussed so far is a C♯-specific re-imagination of the overall design of [Klonatos et al., 2014]. For their query compilation approach, similar to [Freytag and Goodman, 1989], the authors suggest to step-wise transform the high-level implementation of the query operators to specialized low-level C code. Going back to our example database, this implies that the code generator produces the code to execute the standard query operators, as would be the case when evaluating the query using LINQ-to-objects, and then compile it using a specialized C♯ compiler. This compiler then contains steps that, at the moment, are not part of the C♯ compiler like automatically fusing query operators, specializing all generic types and lambda function calls and lowering the implementation to C code. The latter may include replacing managed types and automatic memory management with unmanaged types and manual memory management. Similar to the case for black-box collections, this is possible because the lifetime of all intermediate memory is known at query compile time and the data stored in the data store follows the relational model. The generated C code is then compiled using existing compiler infrastructure and finally executed to evaluate the query.

Re-evaluating the code generation and compilation strategy, it seems unnecessary to go from an expression tree representation of LINQ's standard query operators to a string representation that evaluates the query using LINQ-to-objects just to call a compiler that parses the C♯ string into a tree representation, optimizes and lowers it to native C code in several rewrite steps and then outputs the optimized C code as a string. Instead, since the expression tree representation of the query already represents the C♯ code to call the query operator methods, it could directly be passed to the specialized C♯ compiler, then be compiled using the optimizations discussed above and returned

as C string or, preferably, compiled to IL instructions representing native code and directly forwarded to the just-in-time compiler to compile them into machine code. Note that this is very similar to our code generation and compilation approach with the difference that our approach performs the transformation to C code in the code generator rather than the compiler and we only use the syntax of LINQ-to-objects and produce the query code by emitting code fragments that are hard-coded into the code generator instead of utilizing the full implementation of the standard query operators. The latter, however, has advantages as it removes the clutter in the code generation process to emit and combine code fragments represented as strings and instead uses the implementation provided by the general-purpose query operators.

[Klonatos et al., 2014] do not provide any information on how their underlying data store is represented and how the system deals with SQL queries that create database tables or insert rows from a table. We assume that the system stores and manages all data as in regular database systems. However, to expose the underlying data store to the generic query operators, they have to dynamically generate high-level type definitions for the data stored in each table and instantiate a generic collection type that represents the table. When compiling the high-level query code that operates on this input representation, the compiler has to lower the high-level query code to the unmanaged level of the data store. This requires the compiler to be aware of the collection type that represents a table and its semantics. Based on these assumptions, the similarities with black-box collections become obvious. Other than that both systems target different use cases, the biggest difference between both is that our approach performs the lowering from a high-level to a low-level representation explicitly in the code generator whereas their approach employs a specialized, database-aware compiler to automatically perform the task.

4.5 Evaluation

In this section, we evaluate a prototype implementation of the black-box collection type. Our prototype supports row-wise and columnar data storage. We evaluate black-box collections using the same six TPC-H queries as used in Section 3.6 and compare them against the results of LINQ-to-objects and our approaches evaluated in Section 3.6. As the data store of black-box collections stores data in a relational fashion without supporting references between data elements, all queries on black-box collections utilize explicit join operations to access related elements from different collec-

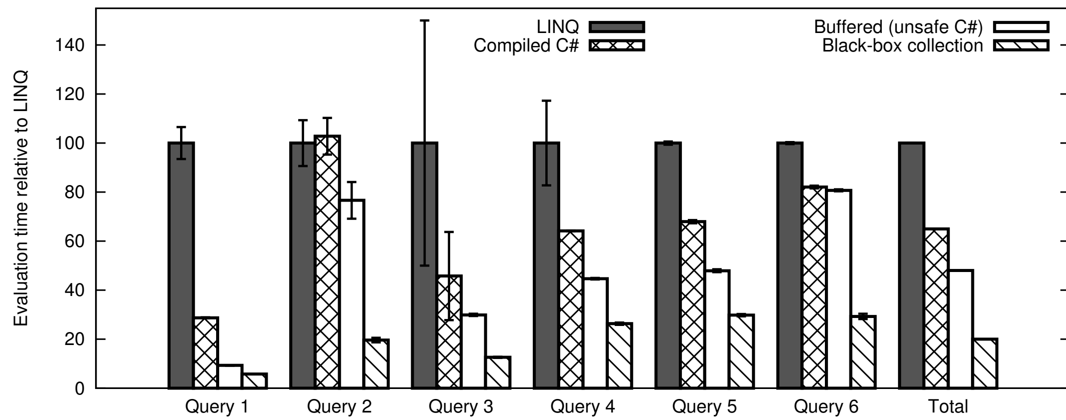


Figure 4.4: Query performance of TPC-H queries 1 to 6 versus a join-based representation of the TPC-H data set

tions. However, in the following, we compare the query performance of black-box collections with the join-based and reference-based object representation from Section 3.6.

Note that all measurements presented here are obtained by loading the data set into the main memory of a freshly started application and then running the benchmark. This results in all managed objects in a collection to be stored in (mostly) consecutive memory addresses in the order they are accessed. However, a real-world application is likely to exhibit a greater degree of fragmentation in the managed heap as typical applications remove and add collection objects, create multitudes of non-collection objects and gradually reclaim memory space of objects that are no longer reachable using garbage collection. This causes queries on collections of managed objects to often perform worse than presented here. In contrast, the performance of black-box collections is independent of allocations or garbage collections in the managed heap. Enumerations are always in the order of memory locations rather than that of an indirection level that may point anywhere in the managed heap. We will have a look at the impact of fragmentation in the managed heap when evaluating self-managed collections in Section 6.6.

We first compare the query performance of data stored in black-box collections with that of data stored in collections of objects. In Figure 4.4, we compare the results of black-box collections with those of objects stored in a join-based representation and in Figure 4.5 with those stored in a referenced-based representation. For data stored as collections of objects, we report the performance of evaluating LINQ queries using LINQ-to-objects, generated C♯ code and generated (`unsafe`) C♯ code

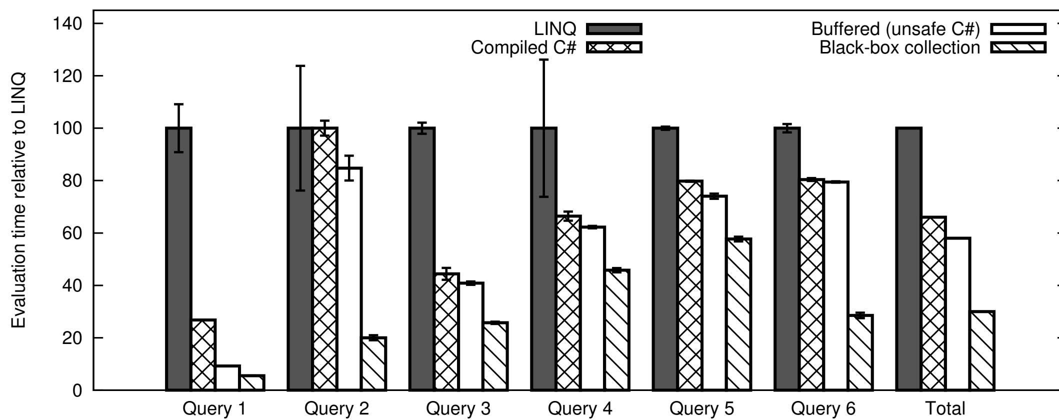


Figure 4.5: Query performance of TPC-H queries 1 to 6 versus a reference-based representation of the TPC-H data set

that stages the data in a single buffer block. In all cases, black-box collections significantly outperform LINQ-to-objects, in some cases even by more than an order of magnitude. Black-box collections also outperform all approaches presented in Section 3.6. It is particularly notable that black-box collections even outperform queries evaluated on the reference-based versions that do not have to perform explicit join operations, whereas black-box collections have to perform explicit joins as they do not support references and there were no index data structures defined on them. The huge performance improvements can be explained by the factors that have been discussed in Section 3.6. Less overhead in the processing model compared to LINQ-to-objects, better performance for `decimal` computations because of direct pointer access to the data elements without the need for staging and less memory allocation overhead because garbage collection is taken out of the equation. Additional benefits come from the vastly improved data layout where consecutive data elements are placed in consecutive memory addresses resulting in better CPU caching and prefetching effects and fewer TLB misses (compared to following references to arbitrary memory locations in the managed heap). Further, data elements are directly accessed in their memory block rather than having to go through an indirection structure (e.g., an array) and data elements do not have any per-object overhead (like managed objects do, e.g., to store the VTABLE).

In Figure 4.6, we compare the query performance of black-box collections stored in a traditional row-wise layout (as used so far) with that of a columnar storage layout. Columnar storage improves query performance of all queries tested. The performance difference between row-wise and columnar storage depends on the query's memory

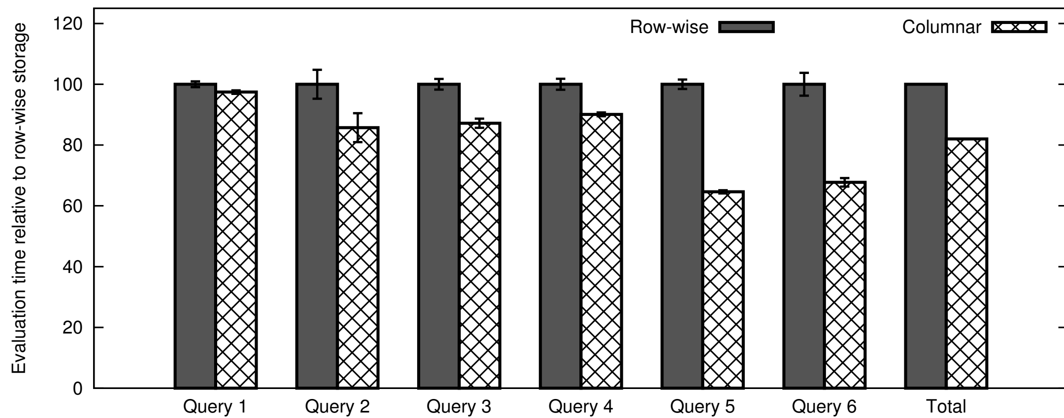


Figure 4.6: Comparison of the query performance of TPC-H queries 1 to 6 between row-wise and columnar data storage

access pattern. The more useful data (i.e., data that is accessed by the query) that can be brought into the CPU with a single cache line fetch, the better the query performance. Queries that on average only touch a small fraction of a database row benefit from columnar storage. On the other hand, if most of each record is touched by the query or the stride in which column values are touched is too big to benefit enough from prefetching, then row-wise layouts perform better. Columnar storage further exhibits a greater CPU cost to compute the memory address of each value within a column. In the benchmarks of Figure 4.6, the selection in query 1 only filters out 3% of all records and the following aggregation touches a big fraction of each record. This characteristic results in query 1 exhibiting the smallest improvement from columnar storage of all queries. Query 6, in contrast, contains four fairly selective queries and, hence, benefits greatly from columnar storage as for most records (5/6th) it is only necessary to touch the column specified in the first selection.

Finally, to put our results into perspective, we compare the query performance of columnar black-box collections with that of a modern commercial database system that incorporates a compressed columnar store to provide fast in-memory OLAP query processing[1]. We store all tables in the database's column store and, in addition, use clustered indexes on the `shipdate` and `orderdate` columns. We believe that these settings are representative for a real-world deployment. We use the `read uncommitted` isolation level and disable parallelized query execution to level the playing field. The LINQ-to-SQL query provider supports the chosen database system to allow to LINQ queries to be translated into SQL queries to be evaluated on the database

[1] We cannot reveal which database we used due to the terms of the licensing agreement.

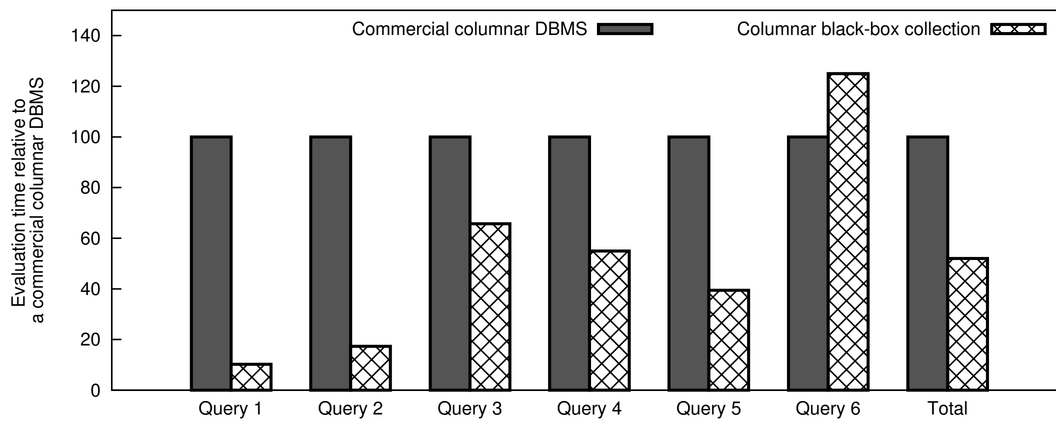


Figure 4.7: Comparison of the query performance of TPC-H queries 1 to 6 between columnar black-box collections and a commercial columnar DBMS

system. However, the presented results are obtained by directly running SQL queries on the database system. The results are shown in Figure 4.7. Black-box collections outperform the database system for all queries but query 6. The performance improvements are in line with what has been shown by related work when comparing compiled query engines with traditional query engines [Diaconu et al., 2013, Klonatos et al., 2014, Neumann, 2011], as is the case for the commercial DBMS. The better performance of query 6 stems from the indexes we defined in the database as two of the selection conditions in query 6 are on `shipdate`. Query 3 also hugely benefits from these indexes.

Chapter 5

Safe manual memory management

5.1 Introduction

This chapter describes the implementation details of a type-safe manual memory management system that constitutes the foundation of what is discussed in Chapter 6 when introducing self-managed collections. The memory management system combines several existing techniques like incarnation numbers, epoch based memory reclamation or indirection to allow type save manual memory management and to enable the characteristics that are required for self-managed collections.

The experiments in Section 4.5 show that black-box collections can significantly improve query processing performance by storing all contained data outside the managed heap. Instead, their data is stored in a second heap of *unmanaged* memory and is managed by the collection type itself. The performance improvements achieved for black-box collections can be attributed to memory management that is aware of the collection's semantics and to allowing the generated query code to use unsafe techniques to process the query. Columnar storage layouts further contribute towards a query processing performance that can level, or even greatly surpass, that of commercial database systems.

However, the approach of Chapter 4 also imposes overheads on the application developer by introducing a relational data store into the managed runtime. Object-relational mapping technologies can unburden the developer of some parts of this overhead, but others remain. These include having to create the initial mappings by annotating class definitions or explicitly having to call `SubmitChanges` to write all the changes on objects returned from the relational data store back. There are also performance overheads that come with relational mappings, e.g., the cost of marshaling data

between both representations, the cost of managing an identity cache and the cost of tracking changes to returned objects.

To address these overheads, we propose self-managed collections. Instead of internally storing relational data, self-managed collections store object-oriented data. By storing objects, self-managed collections can directly return data elements from the data store to the application. There is no need for an object-relational mapping and, hence, for the overheads implied. To maintain the performance characteristics of black-box collections, we propose to store contained objects outside the managed heap and, hence, outside the scope of garbage collection. Similar to what has been the case for black-box collections, self-managed collections manage the storage of contained objects themselves. For this purpose, we introduce a novel type-safe manual memory management system that is designed specifically for use with self-managed collections. In this chapter, we introduce the manual memory system. Self-managed collections will be introduced in Chapter 6. As the memory management system is purpose-built for self-managed collections, we will not evaluate it separately, but evaluate the characteristics relevant for self-managed collections when evaluating them in Section 6.6.

5.2 Design overview

The primary difference between automatically and manually managed types is the requirement that manually managed objects have to be freed manually by the developer using the `free` keyword[1], whereas for automatically managed types, the runtime employs garbage collection to automatically reclaim the memory space of objects that are no longer reachable by the application. By manually freeing objects, there is no guarantee that the application does not hold references to an object after it has been freed (i.e., dangling pointers). Accessing this object violates memory safety. To address this issue, our manual memory management system assumes that all references to manually-managed objects implicitly become `null` when the object is freed. Dereferencing them will throw a `NullReferenceException`.

Another requirement for the manual memory system is to provide type and thread safety. The type safety guarantees for manually managed types are not the same as for automatically managed ones. We guarantee that a reference always refers to an instance of the same type and that this instance is either the one that was assigned to the

[1] Unless they are automatically freed when going out of scope

reference or, if the instance has been freed, `null`. This differs from automatically managed types that guarantee that a reference points to the object it was assigned to for as long as the reference exists and refers to that object. We will introduce the techniques to guarantee type-safety in Section 5.3 and deal with thread-safety in Section 5.4.

As we intend manually managed types to be excluded from garbage collection, we have to impose further restrictions on them. They may not contain references to automatically managed types as, otherwise, they have to be scanned with each garbage collection to check whether the referenced managed objects are alive. However, manually managed types may contain references to other manual managed types and automatically managed types can reference manually managed types. To address the duality of having two class types with different semantics we introduce the `tabular class` keyword to indicate classes backed by the manual memory manager.

As the manual memory management system is purpose-built for the use with self-managed collections, which will be introduced in Chapter 6, its design is based on the following assumptions about self-managed collections:

- Collections contain several millions of objects and, hence, the application maintains minimal references to these objects in any structure other than the collection itself.
- The dominant way of accessing the objects in a collection is through enumeration using language-integrated queries which are automatically compiled into imperative query functions that produce the query's result.
- These queries are predominantly read-only. Their result sets tend to consist of new (managed) objects to be used by the application (e.g., aggregate computations) rather then objects from the base collections. As the result sets usually are part of the application's output to the user (e.g., as tables, graphs or charts), they tend to be rather small.
- The objects stored in a collection are usually updated or removed from the collection by iterating over all collection elements in a LINQ query and performing the updates / removals on the objects returned by the query.

Based on these assumptions, the memory management system is designed to optimize the performance of query functions that are automatically generated from LINQ statements; if necessary even at the expense of other use-cases, e.g., the cost of random accesses to collection objects through references. It is, further, assumed that main

memory is cheap and plentiful and, thus, the design favours performance over space wastage caused by delayed memory reclamations or compactions.

The manual memory management system described in this chapter requires a deep integration into the managed runtime. The `alloc` and `free` methods of the memory system are part of the runtime API and are called by the application to allocate and free manually managed objects. To ensure type-safe reference accesses, references to manually managed objects are implemented as fat pointers that store additional meta data and, before granting access to a manually managed object, the memory management system has to verify the correctness of the access based on the reference's meta data. In contrast, references to automatically managed types are directly translated into pointer-to-memory addresses by the JIT compiler. To integrate the manual memory system into the runtime, the JIT compiler has to be made aware of manually managed type references and the code that has to be produced when dereferencing them.

5.3 Safe manual memory management

The memory manager allocates objects from unmanaged memory blocks, where each block only serves objects of a certain type. Only storing objects of a certain type in each block and disallowing variable-sized objects to be stored in-place ensures that all object headers in a block remain at constant positions within that block, even after freeing objects and reusing their memory space for new ones of the same type. This ensures type-stability for all objects stored by the memory manager. The base address of all memory blocks is aligned to the block size to allow extracting the address of the block that an object is stored in from a pointer to that object. Being able to access an object's memory block from its pointer provides several benefits, one of them is the ability to store type-specific information like VTABLE pointers only once per block rather than with every object. The memory space in a block that is occupied by an object is referred to as the object's *memory slot*. Each memory slot consists of an object header and the object's data, i.e., its non-static fields. Object headers consist of a 32-bit *incarnation number*. Incarnation numbers ensure that objects are no longer accessed after the object was freed. For each memory slot, the incarnation number is initialized to zero and incremented whenever an object is freed. References to objects store the incarnation number of the memory slot together with a pointer to the memory slot. Before accessing an object's data, the system verifies that the incarnation number in the reference matches that in the object's header and only then allows

access to the object. If the application tries to access an object that has been freed, i.e., an object where the incarnation numbers do not match, then the system throws a null reference exception. The JIT compiler automatically injects these checks when dereferencing manually managed objects. The combination of these techniques, in combination with C♯'s existing type-safety guarantees for class types (e.g., no unsafe type casts or pointer arithmetic), guarantees type-safe manual memory management as defined in Section 5.2.

We illustrate the safe reclamation of memory slots using incarnation numbers in Figure 5.1. When allocating the `adam` object, the system allocates it in a new memory slot with the incarnation number initialized to zero. Freeing the `adam` object increments the incarnation number of its memory slot to 1, thereby implicitly setting the `adam` reference that is still held by the application to `null`. The allocation of `tom` reclaims the memory slot previously held by `adam` and uses it to store the `tom` object. The incarnation number stored in the returned reference is the memory slot's current incarnation number 1. At this point, the `adam` and `tom` references both refer to the same memory location, but only `tom`'s data is still accessible as the incarnation number stored in the reference and the memory slots's incarnation number only match for `tom`. Trying to print `adam`'s name will cause a null reference exception.

5.3.1 Incarnation number overflow

Incarnation numbers can overflow, which may cause references to already freed objects to become valid again. To prevent this, the memory system stops reusing memory slots once they reach the maximum incarnation number. If freeing an object sets its incarnation number to the maximum value, the slot becomes a *dead slot* and all references to that slot remain `null` until the overflow is dealt with. Note that each memory slot has an individual incarnation number that is only incremented once with every reuse of that slot for a new object. With 32-bit incarnation numbers, overflows are unlikely to occur in the lifetime of a typical application. However, should incarnation slot overflows occur, then the memory system allows dead slots to accumulate until their number surpasses a threshold. It then invokes a background thread to scan all references in the application and set all invalid references (ones where incarnation numbers do not match) to `null`. This process has similarities with the marking phase of traditional garbage collection, however, does not require any other application processing to be suspended while all references held by the application are scanned as

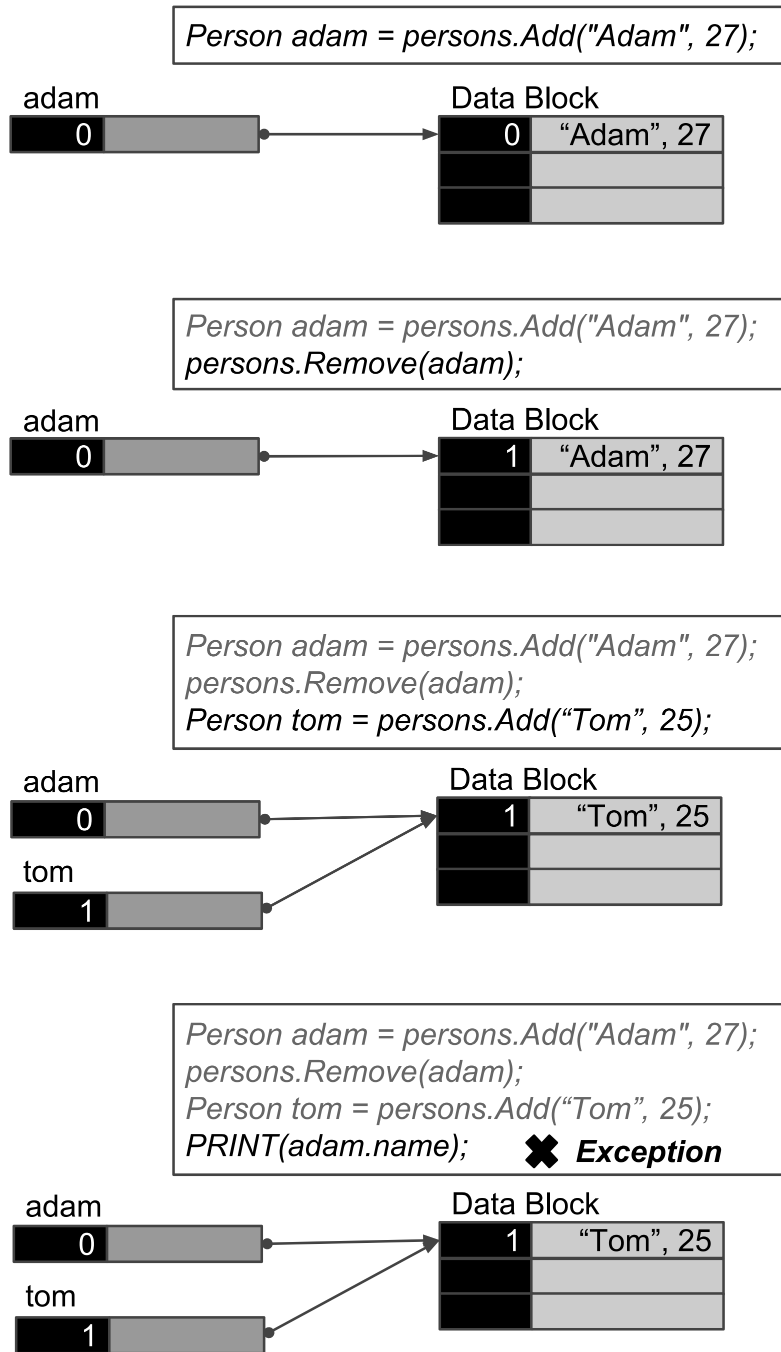


Figure 5.1: Safely reclaiming memory using incarnation numbers

long as the application cannot obtain new references to dead slots. To ensure that this is the case, the memory manager prevents the application from copying existing references to dead slots by, before copying a reference to a manually-managed object, automatically checking that the reference is valid and, otherwise, setting it to `null`. Once the background thread is finished, the incarnation numbers of all dead slots that

Thread 1	Thread 2
`if (CHECK_INC(adam))`	
	`persons.Remove(adam);`
	`Persontom = persons.Add(''Tom'', 25);`
`PRINT(adam.name);`	

Figure 5.2: Concurrency conflict

accumulated before the start of the background thread are reset to allow them to be reused again.

5.3.2 Memory contexts

Objects have so far been grouped into memory blocks with other objects of the same type. However, in many use cases, certain object types exhibit temporal and/or spatial locality: e.g., objects of the same collection are more likely to be accessed in close proximity. By defining *memory contexts*, the memory system allows the programmer to instruct the allocation function to allocate objects in the blocks of a certain context (e.g., a collection). The memory blocks of a context only contain objects of a single type and only the ones that have been allocated in that specific memory context. Memory contexts can be seen as explicit regions, similar to [Gay and Aiken, 1998].

5.4 Concurrency

Incarnation numbers provide memory safety by protecting references from accessing objects that have been freed. However, they do not protect objects from being freed and reused concurrently while being accessed. Consider Figure 5.2: Thread 2 frees and reuses the memory slot referenced by the `adam` reference just after Thread 1 successfully checked the incarnation numbers for the same object. As Thread 1's incarnation number check was successful, the thread accesses the object, which is now no longer `Adam`, but `Tom`. This behavior violates the type-safety requirement of always returning the object assigned to a reference, or `null` if the referenced object has been freed. We refine the requirement for the concurrent case by specifying the check of the incarnation numbers to be the point in time where the requirement must hold. Thus, all accesses to objects are valid as long as the incarnation numbers matched at the time

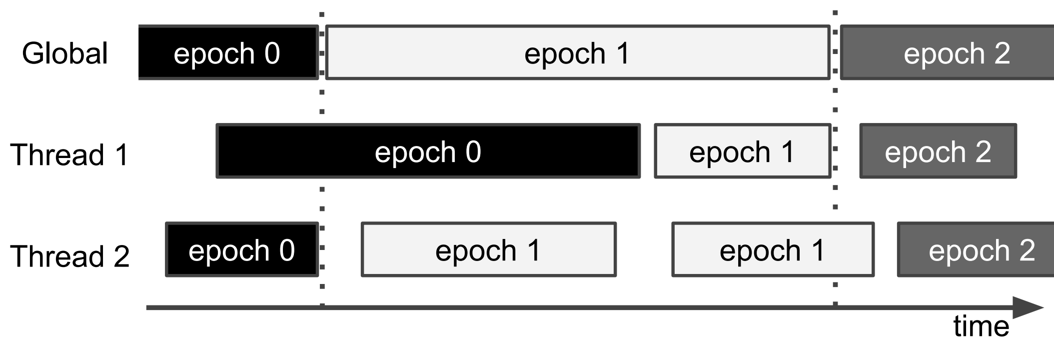


Figure 5.3: Epoch-based memory reclamation

they were checked. To ensure type safety, i.e., that all object accesses are to the same object that the incarnation number check was performed on, the memory system must prevent concurrent threads from overwriting the object (by reclaiming its memory slot) until all threads have finished accessing it. Note that this does not disallow concurrent threads from freeing the object concurrently, but it prevents them from reclaiming the object's memory slot to allocate new objects.

Lock-free concurrent data structures have faced similar issues when reclaiming memory in the absence of garbage collection, with various solutions having been proposed, e.g., [Desnoyers et al., 2012, Fraser, 2004, Michael, 2004]. To prevent the reclamation of memory slots that are still being accessed concurrently, the memory manager employs a variation of epoch-based reclamation [Fraser, 2004]. Epoch-based reclamation is based on *grace periods*. Grace periods are time intervals during which a thread can access objects without fear that the memory space of the object could be concurrently reclaimed. Thus, grace periods are the time interval during which a thread can access objects without re-checking their incarnation numbers to ensure type safety. The length of grace periods can be controlled by varying the frequency in which threads re-check incarnation numbers when accessing the fields of an object. An object's memory slot can only be reclaimed once all threads that may have accessed the object in a grace period have completed this grace period.

Epochs are time intervals during which all threads pass at least one grace period. The memory slot of objects that are freed in some epoch e can safely be reclaimed in epoch $e+2$ because, by that time, all threads that accessed the object in a grace period in epoch e must have completed at least one grace period since accessing the object and, hence, cannot access the object any more. There cannot be any further accesses to the object after epoch e because, by that time, the object is already freed and the incarnation number check will fail. The memory system tracks epochs by

```
1  void enter_critical_section() {
2    global->sectionCtx[threadId].epoch = global->epoch;
3    global->sectionCtx[threadId].inCritical = 1;
4    memory_fence();
5  }
6
7  void exit_critical_section() {
8    memory_fence();
9    global->sectionCtx[threadId].inCritical = 0;
10 }
```

Figure 5.4: Entering and exiting critical sections

maintaining a global epoch. In addition, each thread tracks its thread-local epoch in a variable that is accessible by all other threads. Upon entering a grace period, each thread sets its thread-local epoch to the current global epoch. When leaving the grace period, each thread checks whether all other threads are in the same, the global epoch e. If this is the case, it increments the global epoch to $e+1$. This procedure ensures the invariant that, at any time, all threads can either be in the global epoch e or in $e-1$ and, hence, the memory slots of all objects freed in $e-2$ can safely be reclaimed. We illustrate how threads progress through epochs in Figure 5.3. When exiting the grace period in epoch 0, Thread 2 observes that all threads are in the same, the global, epoch 0 and increments the global epoch to 1. When finishing the first grace period in epoch 1, Thread 2 cannot increment the global epoch to 2 as Thread 1 still is in epoch 0. However, when Thread 1 finishes the grace period in epoch 1, it observes that all threads are in epoch 1 and, hence, sets the global epoch to 2.

Epoch-based reclamation is implemented by performing all accesses to manually managed objects in critical sections (i.e., grace periods). To do so, the JIT compiler automatically injects code to enter and exit critical sections around (reference) accesses to manually managed objects. We outline the code to enter and exit a critical section in Figure 5.4. Upon entering a critical section, each thread sets its local epoch to the current global epoch (line 2) and sets a flag to indicate that the thread is currently in a critical section (line 3). When exiting the critical section, the thread clears this flag (line 9). To ensure that the code to enter a critical section is executed before accessing the object and that there is no further accesses to the object after exiting the critical

```
1  bool try_increment_epoch() {
2    enter_critical_section();
3
4    // Check that all threads are in the same epoch e
5    for (int i = 0; i < global->numThreads; i++) {
6      if (global->sectionCtx[i].epoch != global->epoch
7              && global->sectionContext[i].inCritical) {
8        exit_critical_section();
9        return false;
10   } }
11   memory_barrier();
12
13   // Increment epoch to e + 1
14   global->epoch = global->sectionCtx[threadID].epoch + 1;
15   exit_critical_section();
16
17   return true;
18 }
```

Figure 5.5: Trying to increment the global epoch counter

section, the JIT compiler also has to inject memory fences (lines 4 and 8) to enforce compiler and CPU instruction ordering. Critical sections are not limited to a single reference access. Several accesses can be combined into a single critical section to amortize the overhead of starting and ending critical sections.

The epoch-based memory slot reclamation used by the memory manager differs for the epoch scheme introduced by [Fraser, 2004] in that epochs are not incremented when exiting grace periods but by the allocation function of the memory manager and that global epochs are not incremented *modulo* three but as a continuous counter. These modifications allow the memory manager to lazily reclaim memory slots on demand when allocating objects. We outline the code to increment the global epoch counter in Figure 5.5. Incrementing the global epoch to $e+1$ is only possible if all threads that are currently in a critical section are in the same, the global, epoch e. Performing this operation in a critical section ensures that concurrent threads can only increment the global epoch to $e+1$, but not $e+2$.

So far, memory slots could be in one of two states, free i.e., the slot has never been used before, or occupied i.e., the slot contains object data. Epoch-based reclamation introduces a third state: Memory slots that are not occupied because the contained object has been freed, but that also cannot be reclaimed yet, because two epochs haven't passed since the object was freed. This kind of memory slot is in the following referred to as *limbo slot*. The state of each memory slot is maintained in a contiguous segment of each memory block, the slot directory. The slot directory stores a 32 bit state for each memory slot. It is accessible through the block's header and indexed by the slot's identifier. The state of a memory slot is encoded in two 16 bit parts, the most significant part stores the actual state code (1 for occupied, 0 otherwise) and the least significant part stores additional state information. For limbo slots, this additional information is the timestamp of the earliest global epoch that the slot can be reclaimed.

5.4.1 Freeing objects

We outline the code to free a manually managed object in Figure 5.6. To improve readability, the code sample omits some of the concurrency control mechanisms required. The `free` function takes a reference to the object as argument (line 1). Here, the reference is represented by the value type `ObjRef` which contains the object's incarnation number (`oref.inc`) and a pointer to the object's memory slot (`oref.ptr`). The `free` function first checks that the reference is to an object that has not been freed already by comparing incarnation numbers (line 5). Then, the object's incarnation number is incremented (line 13) to ensure that no concurrent thread can obtain a direct pointer to the memory slot anymore. To ensure that the memory slot of the object cannot be reclaimed until two epochs have passed and, hence, no other thread can still have a direct pointer to the memory slot at the time of reclamation, the state of the memory slot in the slot directory (`blk->slots`) is set to limbo (line 16). This is done by setting its state code to zero and its reclamation epoch to the current global epoch plus two. Finally, the system checks if the block has to be added to the *reclamation queue* (line 20), a queue of same-type memory blocks that qualify for reclamation, along with the earliest epoch when the block can be reclaimed (global epoch plus two).

The memory system tolerates a certain degree of unused memory space due to limbo slots before trying to reclaim their memory space. The *reclamation threshold* regulates the fraction of limbo slots allowed per memory block before attempting to reclaim them. If the `free` function encounters a block that contains more limbo slots

```
1  void free(ObjRef oref)
2  {
3    // Ensure that reference is valid
4    ObjectHeader* ohead = (int*)oref.ptr;
5    if (oref.inc != ohead->inc)
6      throw new NullReferenceException();
7
8    // Get pointer to memory block and slot identifier
9    MemoryBlock* blk = MemoryBlock.GetMemoryBlock(oref.ptr);
10   int slotId = MemoryBlock.GetSlotId(blk, oref.ptr);
11
12   // Increment incarnation number
13   ohead->inc++;
14
15   // Mark slot as limbo and set removal timestamp
16   blk->slots[slotId] = ThreadContext.globalEpoch + 2;
17   blk->numLimbo++;
18
19   // Add block to reclamation queue if necessary
20   if (blk->onReclamationQueue == false &&
21        blk->numLimbo > blk->reclamationThreshold)
22     MemManager.addToReclamationQueue(blk,
23                       ThreadContext.globalEpoch + 2);
24 }
```

Figure 5.6: Freeing an object

than specified by the reclamation threshold, the block is put on the reclamation queue (if it is not already on there) as possible candidate for reclamation. However, the block is only considered for reclamation after two epochs passed. This ensures that, when reclaiming memory slots from the block, the number of limbo slots that can be reclaimed is at least as high as the number specified by the reclamation threshold. This allows to lazily perform memory reclamation on-demand when searching for a memory slot to allocate a new object.

```
1  ObjRef alloc()
2  {
3    // Check if we need to increment the global epoch
4    MemoryBlock* blk = PeekReclamationQueueHead();
5    if (blk != null &&
6         blk->startReclaimingEpoch > ThreadContext.globalEpoch)
7      TryIncrementEpoch();
8
9    // Get memory slot for object
10   AllocCtx ctx = new AllocCtx();
11   GetMemorySlot(&ctx);
12
13   // Set Slot directory to occupied
14   ctx.blk->slots[ctx.slotId] = (1 << 16);
15
16   // Return reference to object
17   byte* ptr = ctx.blk->data + (ctx.slotId * this.objectSize);
18   return new ObjRef() { ptr = ptr, inc = *((int*)ptr) };
19 }
```

Figure 5.7: Allocating a new object

5.4.2 Allocating objects

In Figures 5.7 to 5.9, we illustrate the code to allocate a memory slot for a new object. The `alloc` function (Figure 5.7) first checks if the global epoch needs to be incremented (line 5). This is the case if the first block in the reclamation queue and, hence, all other blocks in the queue, cannot be reclaimed yet because its reclamation epoch is greater than the current global epoch. Then, it finds a memory slot for the new object by calling `GetMemorySlot` (line 11, code in Figure 5.9), sets the memory slot's slot directory entry to occupied (line 14) and finally returns a reference to the manually-managed object (line 18) that contains a pointer to the memory slot and the incarnation number of the object.

`GetMemorySlot` (Figure 5.9) calls `GetMemoryBlock` (Figure 5.8) to get a memory block that may contain free or reclaimable memory slots. We only allow a single thread to claim memory slots from a block at a time (though there can be concurrent free operations on the same block). Before using a block for allocations, the thread has to claim

```
1  void GetMemoryBlock(AllocCtx* ctx)
2  {
3    // If possible, use thread-private block
4    MemoryBlock* blk = GetThreadLocalBlock();
5    if (blk != null) {
6      ctx->blk = blk;
7      return;
8    }
9
10   // Try to claim a blk from the head of the limbo list
11   blk = ClaimBlockFromReclamationQueue();
12   if (blk != null) {
13     AddThreadLocalBlock(blk);
14     ctx->blk = blk;
15     return;
16   }
17
18   // Allocate a new memory block
19   blk = NewMemoryBlock();
20   AddThreadLocalBlock(blk);
21   ctx->blk = blk;
22 }
```

Figure 5.8: Finding a memory block for a new object

the block to prevent any other thread from using it and store it in a thread-local variable to allow several successive allocations of the same type (or memory context) from that block. `GetMemoryBlock` first checks if a thread-local block is already claimed (line 4) and, if this is the case, returns the block. Otherwise, it checks if there is a block on the reclamation queue that qualifies for reclamation (line 11) and, if this is the case, claims the block and returns it. If there is no qualifying block on the reclamation queue, a new block is allocated from unmanaged memory (line 19), claimed by the thread and returned to the caller. Claiming a block from the reclamation queue requires synchronization primitives as other threads may try the same. Claiming a block as thread-local does not require any further synchronization mechanisms.

Memory blocks remain claimed by a thread for the cause of a single iteration

```
1  void GetMemorySlot(AllocCtx* ctx)
2  {
3    do {
4      GetMemoryBlock(ctx);
5      MemoryBlock* blk = ctx->blk;
6      int slotId;
7
8      // First check if there are free slots
9      if (blk->hasFreeSlots) {
10       slotId = blk->lastClaimed + 1;
11       if (slotId <= blk->lastSlot) {
12         blk->lastClaimed = slotId;
13         ctx.slotId = slotId;
14         return;
15     } }
16
17     // Otherwise check if there are limbo slots to reclaim
18     else {
19       for (slotId = blk->lastClaimed + 1; slotId <= lastSlot; slotId++) {
20         if (blk->slots[slotId] <= ThreadContext.globalEpoch) {
21           blk->lastClaimed = slotId;
22           ctx.slotId = slotId;
23           return;
24     } } }
25
26     // Block has no further free slots or reclaimable limbo slots
27     RemoveThreadLocalBlock();
28   } while (true);
29 }
```

Figure 5.9: Finding a memory slot for a new object

over the block's slot directory. This iteration is spread over several successive calls to `GetMemorySlot` and, hence, object allocations. The block's `lastClaimed` variable maintains the progress of the iteration between successive allocations by storing the slot directory position of the last slot that was considered by the previous allocation. It is initialized to -1 when claiming a block. There are two cases for claiming memory slots: if the memory block was freshly allocated, then all allocations are from free slots, otherwise, all allocations are from limbo slots. In the former case, all free slots are in consecutive memory slots and, hence, `GetMemorySlot` always claims the slot

following `lastClaimed` and updates `lastClaimed` accordingly (lines 9 to 15). In the latter case, `GetMemorySlot` iterates over the slots in the slot directory starting with the slot following `lastClaimed` and, for each slot, checks if the slot is a limbo slot that can be reclaimed (lines 19 to 24), i.e., where the state code is 0 and the reclamation epoch is smaller or equal the current global epoch. If such a slot is found, `lastClaimed` is set to this slot and the slot is returned. The number of slots that need to be scanned before finding a limbo slot that can be reclaimed depends on the reclamation threshold. For instance, if blocks can host one hundred objects and are added to the queue once they contain more than 5% limbo slots, then each allocation scans an average of twenty slots to find a reclaimable limbo slot. The actual number is likely to be smaller as additional removals might have happened in the meantime. If the last slot of the block is reached without finding a slot that can be claimed, then the thread removes its claim from the block and repeats the procedure to try to claim a memory slot from a different block.

Instead of using epoch-based memory reclamation, we could have also used hazard pointers [Michael, 2004] to prevent concurrency artifacts. In this case, dereferencing manually managed objects would require the use of a hazard pointer to access the object safely. Hazard pointers require a memory fence after assigning the reference to one of the thread's hazard pointers to prevent compiler and processor instruction reordering and, hence, to enforce that the hazard pointer is set before accessing the reference. We chose epochs over hazard pointers, because epochs allow us to amortize the cost of a memory fence over all object accesses inside a critical section, whereas hazard pointer require a memory fence for each individual memory access. This decision will become more apparent when we discuss how the manual memory management system is used to provide fast query processing for self-managed collections in Chapter 6.

5.5 Concurrent compaction

As the memory manager automatically fills empty slots with newly allocated objects, it does not require any compaction as long as the number of objects of a type (or memory context) remains fairly stable over time. We assume that this is the common case. However, the memory manager behaves poorly if the number of objects of a certain type shrinks heavily. Due to a lack of garbage collection, it is not possible to compact low-occupancy blocks because moving an object to a different memory slot requires to atomically update all references to the object, which in turn requires suspending

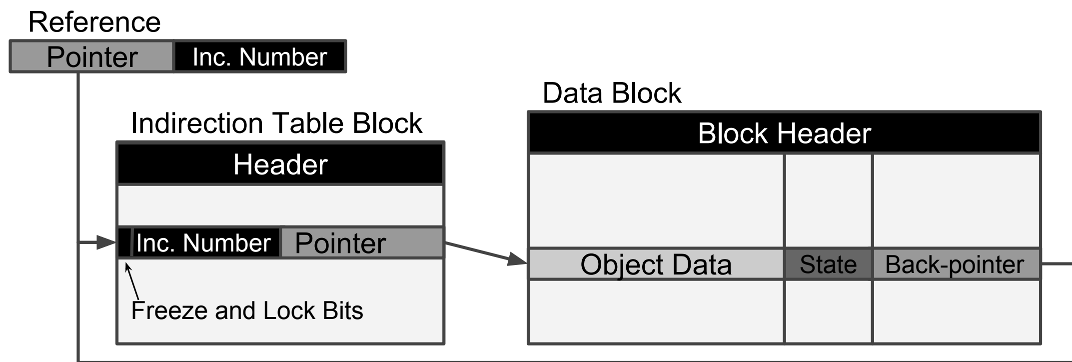


Figure 5.10: Accessing object data through indirection

all application processing in the meantime. Even if there are empty blocks of a certain type, they cannot be freed or reused for a different type because the application may still hold references to the objects that were stored in the block and their incarnation numbers, stored in their memory slot, are needed to ensure that these references are interpreted as `null`. We address these issues by introducing indirection to object references. Indirection is not a new concept for virtual machines. Early virtual machines, e.g., Smalltalk VMs of the 80s [Goldberg and Robson, 1983], used indirection to translate object identifiers to virtual memory addresses to locate objects in memory. However, indirection is no longer used in modern VMs because of the higher cost of memory accesses compared to a direct representation.

The indirection architecture is sketched in Figure 5.10. Instead of pointing to the object's memory slot, object references point to indirection table entries, which in turn point to the memory slot. The incarnation number is no longer stored in the memory slot but moved to the indirection table entry. The indirection table is organized as a linked list of indirection blocks, each storing a fixed number of indirection table entries. As the size of each indirection table entry (96 bits) is independent of the object type, we use a global indirection table to serve objects of any type. This scheme allows empty indirection blocks and empty indirection slots to be reused by a different type and, hence, allows the indirection table to better adapt to a shrinking number of objects of a certain type, as their indirection table blocks and entries can be reused for any other type. Moreover, the space wasted by dead slots (caused by an incarnation number overflow) is limited by the size of an indirection table entry. When moving objects to a different memory slot (e.g., as part of a compaction), the memory manager requires access to an object's indirection table entry from its memory slot. To allow this, each memory block contains another consecutive memory area, referred to as

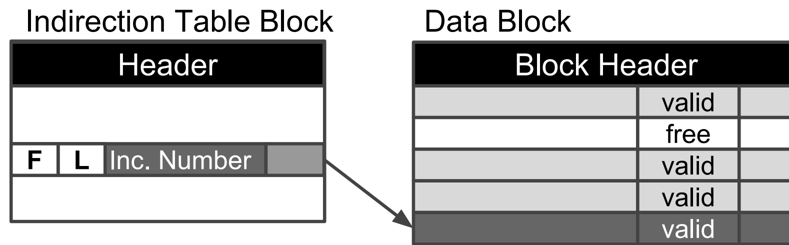
back-pointer in Figure 5.10, that stores pointers to the indirection table entries of all objects stored in the memory block. As is the case for the slot directory, each element in the back-pointer array is indexed using the corresponding memory slot's identifier.

As shown in Figure 5.10, each data block is divided into four consecutive segments: block header, object store, slot directory, and back-pointers. The object store contains all object data. Each object's data is accessible through a pointer from the corresponding indirection table entry or through the identifier of the object's slot in the memory block. The slot directory stores the state of each slot and additional state-related information (for a total of 32 bits). The back-pointer stores a pointer to the object's indirection table entry. The slot directory entry and the back-pointer are accessible using the object's slot identifier.

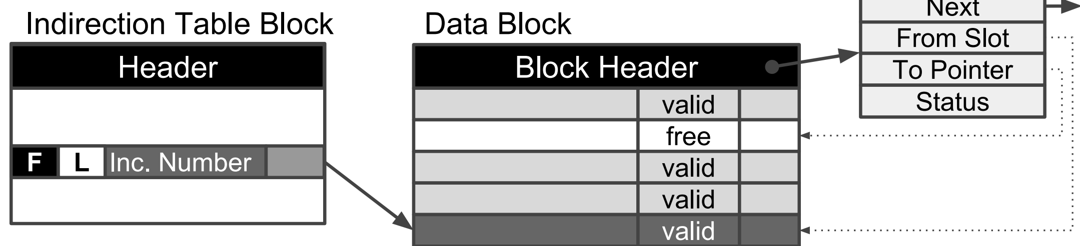
The indirection table allows to move objects within and across memory blocks (e.g., for compaction) without having to update all references held by the application. Atomically updating the pointer in the indirection table suffices to ensure that all threads use the new location of the object when accessing the object's memory slot through its indirection table entry. However, threads that already have obtained a direct pointer to the old memory location through the indirection table may cause inconsistencies by performing updates on an outdated memory location. As we intend to address these inconsistencies without having to suspend all application processing (as would be the case for garbage collection), we extend the epoch scheme described in Section 5.4 for object relocation. We reserve the two most significant bits of the incarnation number in the indirection table for a *frozen* flag [Braginsky and Petrank, 2011] and a *lock* flag. The new epoch scheme requires extending the `try_increment_epoch` function (from Figure 5.5). After a thread successfully increments the global epoch, it checks if a compaction is necessary. The global epoch cannot be incremented in the meantime because the thread is still in a critical section using the previous epoch. If compaction is necessary the thread sets the global `nextRelocationEpoch` to $e+2$ (e is the thread-local epoch and $e+1$ is the global epoch that was just incremented) and then awakes the compaction thread. We use a separate thread for compaction to ensure that no application thread is blocked for the duration of the compaction. Once the relocation epoch is set, no other but the compaction thread can increment the global epoch until the compaction is finished (epoch $e+3$). To guarantee this, the compaction thread is run in a critical section that uses the thread-local epoch e, which prevents all other threads from incrementing the global epoch.

The compaction thread is active through two epochs: the freezing epoch $e+1$ and

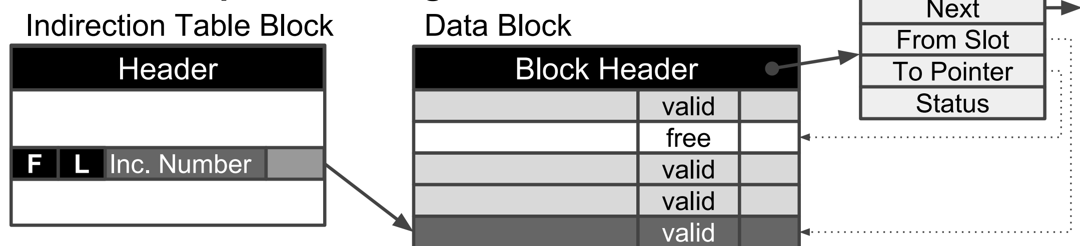
Before:

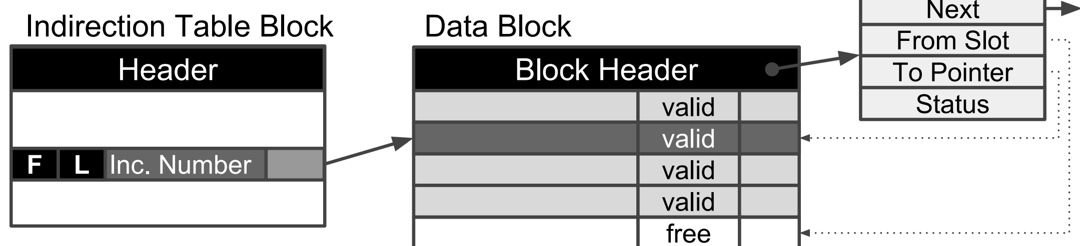


Freezing Epoch:



Relocation Epoch, Moving Phase:





After:

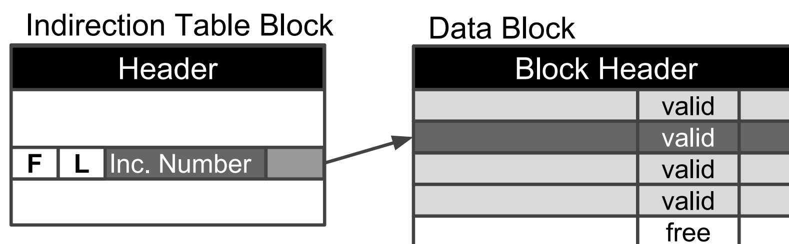


Figure 5.11: Relocating an object

```
1  void* dereference_object(ObjRef oref) {
2    if(oref.inc == oref.ptr->inc) {
3      return oref.ptr->memptr;
4    } else if (oref.inc == (oref.ptr->inc & FL_MASK)) {
5
6      // First case:
7      if (global->sectionCtx[threadID].epoch
8              != global->nextRelocationEpoch) {
9        return oref.ptr->memptr;
10
11     // Second case:
12     } else if (!global->inMovingPhase) {
13       bail_out_relocation(oref);
14       return oref.ptr->memptr;
15
16     // Third case:
17     } else {
18       relocate_object(oref);
19       return oref.ptr->memptr;
20     }
21   } else {
22     throw new NullReferenceException();
23 } }
```

Figure 5.12: Handling all cases when dereferecing an object

the relocation epoch $e+2$. In the freezing epoch it iterates over all blocks that need compaction (marked by previous allocations/removals). For each block, it constructs a list of all objects that have to be moved. For each object scheduled for relocation, the list contains the identifier of its old memory slot and a pointer to the new memory slot it will be moved to. This list is accessible through the block's header. The thread then sets the frozen bit in the indirection table entry of each slot that is scheduled to be copied[2]. Once all blocks are prepared for compaction, the thread waits until all other threads are in the freezing epoch $(e+1)$ and then increments the global epoch to $e+2$ to start the relocation epoch. The relocation epoch consists of two phases: the

[2]By using a CAS operation; this requires `free` to also use CAS to increment incarnation numbers

waiting phase, which lasts until the compaction thread observes that all other threads are in the relocation epoch, and the moving phase that starts thereafter. While waiting, the compaction threat continuously tries to increment the global epoch to proceed to the moving phase. Once in the moving phase the compaction thread makes this phase globally visible by setting a global variable to indicate that frozen objects may now be moved. It then iterates over all blocks scheduled for compaction. For every slot to be moved, it atomically locks the incarnation number by setting the lock bit and copies the object to the new location, updates the pointer in the indirection table, unsets the lock and frozen bits, and marks the relocation as successful in the block's relocation list. Once all scheduled relocations are done, the compaction thread increments the global epoch to $e+3$ (all threads are guaranteed to be at $e+2$ at this point), exits its critical section to allow other threads to increment the global epoch, and goes back to sleep. We illustrate the steps to move an object inside a memory block in Figure 5.11.

The frozen and lock flags in the incarnation number have to be dealt with when accessing an object through a reference. If the frozen flag is not set (and, hence, also the lock flag), then there is no risk of the object being moved in the current epoch, no matter whether the thread is in a regular, a freezing or a relocation epoch. In this case, it is sufficient to compare the incarnation number stored in the reference with the one in the indirection table entry as has been the case before. If the object has not been freed since the reference was acquired, it is safe to access the object's memory slot. Note that, as relocations are rare, this is the common case and there is no additional overhead posed by possible relocations. However, if the frozen flag is set in the incarnation number (i.e., the first incarnation number comparison fails, but a second that excludes frozen and lock bits succeeds), there are three cases (as outlined in Figure 5.12):

(1) The thread is in the freezing epoch. There will not be any relocation in this epoch, so it can safely access the memory slot of the object (lines 7 to 9).

(2) The thread is in the waiting phase of the relocation epoch and not all threads are in the relocation epoch yet. A relocation might happen while the thread accesses the object so it cannot proceed. However, it also cannot relocate the object because not all threads are in the relocation phase so they do not expect relocations yet. The only option is to bail out from relocating the object. To do so, the thread finds the object's entry in the block's relocation list, atomically sets the lock bit in the object's incarnation number, then sets the status of the relocation to failed (in the object's relocation list entry), and finally unsets the frozen and lock bits. If the

lock bit has already been set by another thread, the thread spins until the lock bit is unset and then rechecks the object's incarnation number. Once the frozen bit is removed, the object's memory slot can be safely accessed (lines 12 to 14).

(3) The thread is in the moving phase of the relocation epoch and all other threads are also in the relocation epoch. The thread again cannot proceed because the object may be moved at any time, but it can help the compaction thread move the object to its new location and then proceed. To do so, it finds the object's entry in the block's relocation list, atomically sets the lock bit in the object's incarnation number and moves the object to its new location. It then sets the status of the relocation to succeeded, and unsets the frozen and lock bits. As in the previous case, the thread spins while the incarnation number is locked by another thread, then rechecks the incarnation number and finally accesses the object's memory slot once the frozen bit is unset (lines 17 to 20).

If the object access is known to be read-only, the thread can always use the original location of the object in the waiting phase of the relocation epoch as its memory slot cannot be reclaimed while it is accessed. This allows the thread to proceed without having to fail the relocation. At the same time, the read access is guaranteed to access the most recent data while in the waiting phase and, if the relocation phase starts while accessing the object, the most recent data at the end of the waiting phase.

When the compaction thread starts iterating over the blocks to be compacted (i.e., the moving phase of the relocation epoch), all failed relocations are visible so the thread can deal with them. If necessary, it extends compaction by one additional epoch to try all unsuccessful relocations again by adding another freezing phase at the end of the relocation epoch and setting the following epoch to be a relocation epoch before exiting the current relocation epoch.

Chapter 6

Self-managed collections

6.1 Introduction

In Chapter 5, we introduced a safe manual memory management system that is purpose-build for self-managed collections (SMCs). In this chapter, we introduce self-managed collections and show how they achieve fast query processing performance by utilizing the manual memory management system. Many of the goals for self-managed collections remain the same as for black-box collections. We want to improve query and application performance for query-intensive applications that store and process huge volumes of database-like data in the memory space of the application. To achieve this, we have to address the inefficiencies that come with LINQ-to-objects, in particular its execution model, as described in Section 3.1 and the inefficiencies that come with garbage collection managed data representations as outlined in Section 4.1. Compared to black-box collections, self-managed collections are intended to allow a deeper integration of the data store and the application by storing collection data as manually managed objects and allowing the application to directly access and modify collection objects using references. The foundations of achieving these characteristics is the manual memory management system introduced in Chapter 5 and query compilation to allow queries to directly operate on the memory blocks of the memory management system rather than on some indirection data structure.

6.1.1 Collection semantics

In order to achieve our goal of fast query processing and improved scalability of query-intensive applications that store huge volumes of data in main memory, we have to

eschew garbage collection and use off-heap storage managed by a collection-specific memory manager. The semantics of conventional collections do not allow them to manage the lifetime of contained objects as these objects may reside outside the collection or be contained in several collections at the same time. To allow self-managed collections to manage all objects stored therein, we introduce new semantics that couple the lifetime of objects to their existence within the collection.

Self-managed collections own their objects. Objects are not created individually by using the `new` keyword. Rather, they are created when they are inserted into the collection and their lifetime ends with their removal from the collection. This accurately models many use cases, as objects often are not relevant to the application once they are removed from their host collection. Consider, for example, a collection that stores products sold by a company. Removing a product from the `products` collection usually means that the product is no longer relevant to any other part of the application. Managed applications, on the other hand, keep objects alive so long as they are still referenced. Object containment is inspired by the table type of relational database systems, where removing a record from a table entirely removes the record from the database.

The following code excerpt illustrates how the `Add` and `Remove` methods of self-managed collections are used:

```
Collection<Person> persons = new Collection<Person>();
Person adam = persons.Add("Adam", 27);
/* ... */
persons.Remove(adam);
```

The collection's `Add` method substitutes the `new` keyword and allocates memory for the object, calls the object's constructor and returns a reference to the object. As the lifetime of each object in the collection is defined by its containment in the collection, mapping the collection's `Add` and `Remove` methods to the `alloc` and `free` methods of the underlying manual memory manager is straightforward. The manual memory management system prevents accesses to the object after it has been removed from the collection by throwing a `NullReferenceException`.

Self-managed collections expose bag semantics. Bags are collections of unordered data that may contain duplicate elements. This ensures that query functions do not need to follow a specific order when iterating over a collection's objects which allows the query function to process objects in memory order. Doing so, in combination

with allocating all objects stored in the collection using a private memory context, better exploits spatial locality and, as a result, improves query processing performance. Iteration is assumed to be the dominant access method to self-managed collections. Indexed access (e.g., `persons[5]`) is *not* supported; but can be manually implemented using references. However, high enumeration performance is favoured over high performance when randomly accessing self-managed objects through references.

As self-managed collections utilize the manual memory management system introduced in Chapter 5, the objects stored in the collection are exposed to the same restrictions as discussed in Section 5.2. In the following, we refer to manually managed objects that are stored in self-managed collections as self-managed objects. To allow the collection to automatically manage contained objects and provide fast enumeration performance, there are some additional restriction for objects stored in self-managed collections. References contained in self-managed objects may either refer to other manually managed objects stored in self-managed collections or ones that are not stored in self-managed collections. As self-managed collections are intended to be well integrated into managed runtimes and should not require the application developer to directly deal with the manual memory management system, the latter are assumed to be owned by the self-managed object referencing them and, hence, have the same lifetime. This implies that all referenced manually managed objects that are not contained in self-managed collections are freed together with the self-managed object that references them. In our case, fixed and variable-length strings are the only allowed manually managed types that may be referenced by self-managed objects but not be part of a self-managed collection themselves. To enforce that manually managed strings have the same lifetime as the referencing self-managed objects, each self-managed object either references private copies of all referenced strings or it references shared copies that are managed using reference counting. In the former case, the private copies are automatically created in the constructor or on assignment, are automatically freed when the referencing self-managed object is freed and their memory space is reclaimed when the self-managed object's memory space is reclaimed. In both cases, overwriting an existing string reference requires adapting its lifetime first. As manually managed strings are part of the referencing self-managed object, we do not allow the application to hold direct references to them but, instead, access them through the self-managed objects. Manually managed types support inheritance and implement interfaces just as normal class types. However, self-managed collections may not be defined on base classes or interfaces. This ensures that all objects in a collection have the same size

and memory layout. Alternatively, self-managed collections could allow base classes and interfaces by maintaining a separate memory context for each type implementing or inheriting from the collection's type. Queries then have to also iterate over all these memory contexts to process the query.

6.1.2 Integration into the managed runtime

Self-managed collections are defined as a class library. However, their underlying manual memory management system requires deeper integration into the managed runtime as described in Section 5.2. This includes to provide self-managed collections access to the manual memory manager's `alloc` and `free` functions and to automatically expand code fragments that dereference manually managed objects returned from a self-managed collection into code fragments that access the indirection table, compare incarnation numbers and deal with not matching incarnation numbers appropriately. Replacing code fragments when dereferencing references to manually managed objects could either be performed in a C♯-to-C♯ preprocessor, the C♯ compiler or the JIT compiler. In the prototype implementation of self-managed collections that is evaluated in Section 6.6, the code fragments are replaced by hand. The code generator and query compiler is implemented as a LINQ query provider as described in Section 3.3. However, under the assumption that LINQ queries typically are statically defined in the source code with only some query parameters (e.g., a constant in a selection predicate) dynamically assigned, it might be preferable to use a C♯-to-C♯ compiler (or the C♯ compiler) to automatically generate specialized query functions for LINQ queries on self-managed collections at compile time. This would eliminate the code generation and compilation overhead at runtime for all LINQ queries that are statically defined in the source code.

6.2 The basic collection type

Similar to black-box collections, self-managed collections aspire to improve the performance of query-intensive applications that store huge volumes of database-like data in their memory space in two areas. First, self-managed collections improve the query performance of language-integrated queries by using query compilation and by providing a data layout that exposes better spatial locality. Second, they improve the overall application performance and scalability by excluding the collection's data from the

managed heap and, hence, the garbage collector.

Improved enumeration performance is achieved by allowing the collection to manage contained objects itself rather than relying on independent systems like garbage collection. To manage contained objects, self-managed collections employ the novel collection semantics introduced in Section 6.1.1 and the manual memory management system as described in Chapter 5. Combining memory and collection management enables the collection to place objects in memory based on the order in which the objects are touched when enumerating over the collection's content in a query. This improves the locality of memory accesses during enumeration, leading to improved query performance compared to iterating over the collection's content through references in an indirection structure that may point anywhere in the managed heap, as is the case for conventional .NET collections. By using bag-semantics, there is no predefined enumeration order on objects in self-managed collections and, hence, they can be placed in memory in any order while still allowing queries to enumerate the collection's content in memory order to provide fast enumeration performance. To further improve locality, each collection contains a private memory context that enables the collection to store all contained objects in memory blocks private to the collection. By placing collection objects in consecutive memory locations and enumerating them in memory order, queries on self-managed collections make better use of cache line and CPU prefetching and reduce the likeliness of TLB misses.

When facing huge volumes of data stored in the memory space of the application, self-managed collections improve the overall performance and scalability of the entire application by excluding this data from garbage collection. Full garbage collections have to scan the entire managed heap to find objects that are no longer reachable by the application and to reclaim their memory. The duration of a full garbage collection, therefore, depends on the size of the managed heap. Storing huge volumes of data in the managed heap (i.e., gigabytes or even terabytes of data) can drastically increase the duration of a full garbage collection. For non-concurrent garbage collectors that require to suspend all application processing for the duration of the garbage collection, this has a significant negative impact on the overall application performance and response time. As concurrent garbage collectors perform big parts of full garbage collections on a background thread, they do not suffer big response time hits when facing increasing volumes of data. However, as the background thread has to be active for a longer period of time, increasing data volumes also affect the overall performance and scalability of the application.

To exclude collection data from the managed heap, self-managed collections have to manage the memory space of contained objects themselves. The collection semantics of self-managed collections, which were introduced in Section 6.1.1, allow the collection to automatically manage the memory space of contained objects by restricting their lifetime to their containment in the collection. As a result, the collection's `Add` and `Remove` methods are merely wrappers around the underlying manual memory manager's `alloc` and `free` functions. Supplying both operation with the collection's private memory context is sufficient to ensure that collection objects are stored in close proximity to improve enumeration performance. As self-managed collections expose bag semantics, we do not need to be concerned about the order in which the memory manager places objects in the context's memory blocks as long as the context will allow directly enumerating over contained objects in memory order. In addition to allocating memory for the object, the `Add` method calls the object's constructor and returns a reference to the object (referred to as `ObjRef`). The `Remove` method calls the `free` function on the supplied object reference.

Fast query processing performance is achieved through the private memory context that each collection uses to allocate and free contained objects. The memory context provides the collection with full access to all unmanaged memory blocks that are used to store the objects contained in the collection. To allow our code generator to produce fast query code from LINQ queries on self-managed collection, we also provide the generated query functions with access to the collection's memory context. This allows the code generator to produce query code that directly operates on the memory blocks that store the objects contained in the collection instead of having to process them through some means of indirection, as is the case for managed collections. Doing so improves query performance by iterating over the collection's objects in their memory order and, therefore, better exploiting spatial locality. As all data is stored in unmanaged memory, query performance can be further improved by accessing collection objects using C-style pointers and using query-specific memory management for intermediate query data as discussed in Chapter 4. The following outlines the generated code for a simple compiled query that enumerates over all objects in the collection by iterating over all valid slots in all blocks in the collection's memory context, checking a predicate on the `age` field, and returning references to all qualifying objects:

```
enter_critical_section();
foreach (Block* blk in persons.GetMemoryContext())
  foreach (Slot i in blk)
```

```
  if (blk->slots[i] == VALID)
    if (blk->data[i].age > 50)
      yield new ObjRef { ptr = blk->backptr[i],
                         inc = blk->backptr[i]->inc };
exit_critical_section();
```

The query uses the memory block's slot directory `blk->slots` to check whether the corresponding memory slot is occupied (in contrast to a free or limbo slot). As the slot directory is stored in a consecutive memory area inside each block, separate from the memory area that stores object data, and its elements are only four bytes wide, it is fairly cheap to iterate over the slot directory to check for valid slots. The query touches the object's data only if the slot is valid. If the slot also satisfies the selection predicate, the query returns a reference to the object. To do so, it uses the back-pointer field `blk->backptr` to obtain a pointer to the corresponding indirection table entry. The reference contains this pointer and the current incarnation number of the object to ensure that the memory slot can safely be reclaimed once the object is removed from the collection. To generate code for more complex queries we follow a similar strategy as described in the previous chapters and utilized in other query compilation approaches, e.g., [Krikellas et al., 2010, Murray et al., 2011, Neumann, 2011].

As the generated query code directly accesses the memory slots of collection objects, it is imperative that the accessed objects cannot be removed from the collection and their memory slot reclaimed while accessing the object. To ensure this, all direct access to memory slots have to be in critical sections. This not only applies to accessing objects in the primary collection(s) of the query, i.e., the one(s) that the query enumerates over, but also objects in other self-managed collections that the query accesses by following references from objects stored in the primary collection(s). To amortize the cost of entering and exiting a critical section around each object access, the query remains in the same critical section either for its entire duration, or for the duration of processing a single memory block. The query compiler chooses the desired granularity for each query based on the requirements of the query. Staying in the same critical section for the duration of the query allows to generate code that stores direct pointers to memory slots of collection objects in intermediate results and data structures. Otherwise the query may only use object references (`ObjRef`). However, it also increases the time until the memory manager can increase the global epoch to reclaim limbo slots. As LINQ queries are lazily evaluated, we enforce that critical sections are exited before a result object is returned and, hence, control is returned to the application. Since most

queries contain several blocking operations (e.g., aggregation or sorting), most of the query processing can still be performed in a single critical sections. Whether objects that are concurrently added or removed from the collection are reflected in the query's result depends on the order in which the query accesses the objects and the concurrent operation modifies the slot directory (if the object is part of a primary collection of the query) or reference / incarnation number (if the object is part of a secondary collection and was reached through a reference). Note that self-managed collections use a lower isolation level than typical database systems, but comparable to other managed collections. Supporting more restrictive isolation levels is not the focus of this work.

6.3 Concurrent compaction

Common uses of self-managed collections do not cause them to shrink significantly; they stay at a stable size or steadily grow. But in the case of rapid shrinkage, we use the relocation technique outlined in Section 5.5 to ensure that all memory accesses through references remain correct. However, the memory management system of Section 5.5 does not guard against queries directly accessing memory slots through enumeration while some blocks of the collection are being compacted concurrently. This may result in inconsistencies where the query misses some objects because they are concurrently being relocated or includes them twice. To prevent this, the compaction scheme of Section 5.5 has to be adapted to also ensure consistent object accesses from queries directly iterating over the memory slots of the collection.

As a first step to avoid inconsistencies in queries that are (partially) evaluated during a relocation epoch, we refine the relation between memory blocks that take part in a relocation. Relocations are triggered by a shrinking number of objects in a collection and serve to improve the memory footprint of the collection by moving objects from some of the low-occupancy blocks to other low-occupancy blocks and, thereby, emptying the former and allowing them to be reclaimed. If, for example, relocation is only triggered once a certain number of blocks have over a threshold, say 33%, of unreclaimed memory slots, then compaction can free one out of three blocks by moving all objects stored in one of the block to the other two. As this scheme results in a small number of blocks that exchange objects with one another, we define compaction groups where each group only contains blocks that exchange objects with each other in the current compaction. In our example, each group contains three memory blocks. Using compaction groups reduces the problem of having to prevent inconsistencies

caused by relocations while processing a query to only having to prevent inconsistencies while processing the blocks contained in a compaction group. To allow the query to process blocks in the same compaction group together, we add a pointer field to the memory block header that states whether the block is part of a compaction group and, if it is, what compaction group the block is in.

As long as a thread is outside a relocation epoch while processing the blocks of the same compaction group, it can access their content as per usual without fear of inconsistent accesses due to concurrent relocations as relocations cannot start until all threads are in the relocation epoch. To ensure this, all blocks of a compaction group are processed in the same thread-local epoch. However, as soon as the compaction group is processed in the relocation epoch, the query has to ensure that concurrent relocations cannot cause inconsistencies. If a query starts processing the blocks of a group in the relocation epoch, we process them differently dependent on whether processing starts during the waiting or the moving phase. In the latter case, the query first performs all relocations in the compaction group and only then processes the entire group using the new locations of all contained objects, hence, processing the objects in their post-relocation memory order.

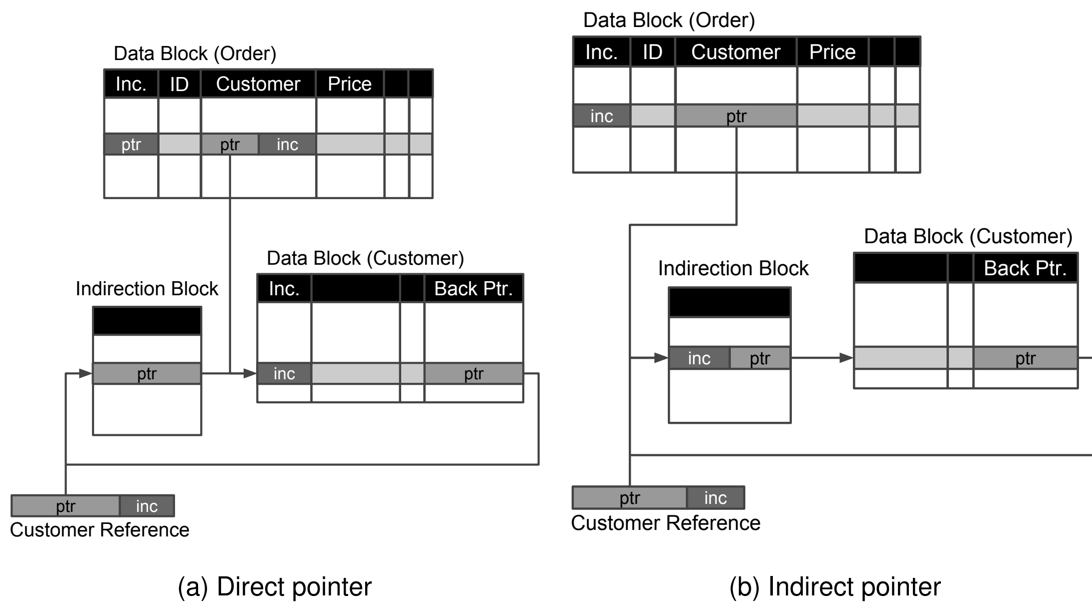
When we encounter a block that is part of a compaction group while we are in the waiting phase of the relocation epoch, we cannot start the relocation because other threads may still be in the previous epoch and, hence, do not expect relocations. In this case, we put the compaction group on a list of groups that still have to be processed, if it is not already on there, and continue the iteration with the next block. After all blocks that do not require compaction are processed, we start processing the compaction groups. If the moving phase has already started, we relocate all objects in a group and then process them in the post-relocation order. Otherwise, we have to process the objects without first relocating them based on their order before performing the relocation. However, processing the compaction group as usual would cause inconsistencies because relocations can start at any time and modify the slot directory, thereby preventing us from using the pre-relocation order. For this reason, queries that start processing a compaction group in the waiting phase of the relocation epoch have to atomically increment the query counter in the compaction group before processing the group and decrement it after they finished processing it. Any relocations performed in the compaction group have to wait for the counter to become zero. Note that relocations only occur in the moving phase of the relocation epoch and, hence, once a relocation waits for the query counter of a compaction group to become zero, there are

no more queries incrementing it. This makes the starvation of the relocation thread impossible. The impact of compaction on query performance can be assumed to be minimal as all overhead is amortized over all objects in a compaction group. Note that, in contrast to garbage collections, compactions are assumed to be very rare and, when they occur, they typically only have to scan the blocks that require compaction (i.e., under-populated blocks) of a single collection. The consistency of memory accesses when following references between self-managed objects during query processing is already ensured by the relocation scheme as described in Section 5.5.

As LINQ queries are lazily evaluated, it is possible that a query processes part of a compaction group, returns an object from the group to the application and then stops processing the query, e.g., using a `break` or `return` statement. In this case, the query would not decrement the query counter in the compaction group and, hence, infinitely stall any thread attempting to relocate objects in the group. To prevent this, we store the current compaction group in a thread local variable before returning control to the application. Furthermore, we automatically modify the code in the loop over the query result that exits query evaluation early to first decrement the query counter of the compaction group stored in the thread-local variable (if any). In addition, we limit the time that an application thread has to wait for a query to decrement the compaction group counter before it can relocate an object to ensure that the response time of the application stays within bounds. As soon as the relocation surpasses this limit, the thread bails out of relocating the objects as described in Section 5.5.

6.4 Direct pointer

Compared to managed collections, self-managed collections exhibit a higher cost when following references to related objects because self-managed collections have to look up the pointer of the related object in the indirection table. For queries that require following references from objects in the primary collection(s), this cost may adversely affect the query performance compared to that of managed collections. For self-managed collections, each dereferencing not only has to check incarnation numbers, but, more importantly, it also has to pay for an additional (random) memory access to the indirection table. We now provide an alternative implementation that solves this problem. We keep indirection for all external references, i.e., references held by the application or managed objects, but, for references between self-managed collections, we store the direct pointer to the corresponding memory slot. To be able to check incarnation



(a) Direct pointer (b) Indirect pointer

Figure 6.1: External and internal pointer representations

numbers in both cases, the incarnation number of a memory slot is moved back into the memory slot (object header) instead of the indirection table. In Figure 6.1, we illustrate the new layout and compare it with the indirection based layout used preciously. The new layout allows queries to directly access referenced objects instead of having to rely on indirection and, hence, improves the query performance for queries that access several self-managed collections by following references from the primary collection(s).

However, when relocating an object, the new memory location of the object has to be updated in the indirection table as well as in all self-managed objects that reference it, which is no longer an atomic operation. We address this by adding a third flag to the incarnation number, the *forwarding* flag. The forwarding flag instructs to use the object's new memory slot instead of the one that has been reached by following the reference. To access the new memory slot of the object, the thread has to use its back-pointer to obtain a pointer to its indirection table entry and then access it to get a pointer to the new memory slot of the object.

Relocating an object, independent of whether the relocation is performed by the compaction thread or an application thread that encountered a frozen incarnation number in the moving phase of the relocation epoch, is mostly performed as described in Section 5.5. Figure 6.2 illustrates the relocation process. The only difference is that the thread performing the relocation now sets the forwarding bit in the incarnation number

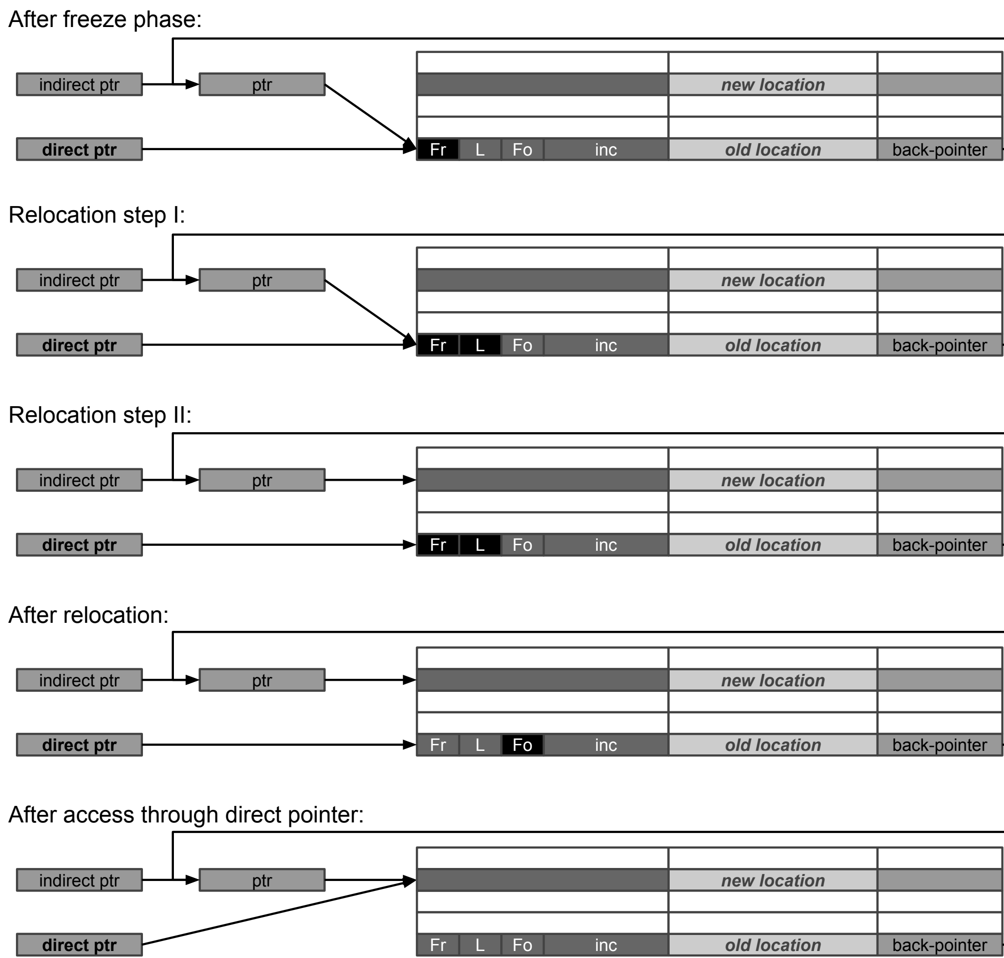


Figure 6.2: Handling relocation with direct pointers

when unsetting the frozen and lock bits after finishing the relocation. Note that, by the time the forwarding flag is set, the relocation is already completed and the new memory location of the object is the only valid one. At this time, it is impossible for accesses through external (i.e., indirect) references to reach the old memory location of the object as the memory slot is accessed through indirection and the indirection table entry already points to the new location of the object. However, direct pointers still point to the old locations.

When following a direct pointer to such a memory slot, then the first incarnation number comparison will fail because the forwarding bit is set, but a second one that excludes the forwarding flag succeeds (i.e., the forwarding flag is set and incarnation numbers match). If this is the case, the thread uses the back-pointer to obtain a pointer to the object's indirection table entry and proceeds with the pointer to the new memory slot of the object stored in the indirection table entry. Threads that follow a direct

pointer to a memory slot that have the forwarding flag set, automatically update the pointer they have followed to point to the new memory slot of the object to improve the performance of future accesses through the same pointer. This is illustrated in the last step of Figure 6.2. Note that any additional overhead is only exposed on objects that have been relocated and, hence, have the forwarding flag set. There is no additional performance cost outside relocations. The extra checks for handling the forwarding flag are performed whenever following a reference between two self-managed objects and the incarnation number comparison fails. This applies to automatically generated query code as well as reference accesses between self-managed objects in the application itself. In the latter case, the JIT-compiler must automatically add the required incarnation number checks. Queries iterating over their primary memory blocks are not affected by direct pointers and, hence, the strategy to handle concurrent compactions, as discussed in Section 6.3, remains unaffected as well. For queries that follow direct pointers to secondary collections, the code generator includes the code to perform the necessary incarnation number checks and handle relocations.

Memory slots that have the forwarding flag set cannot be reclaimed because there may still be direct pointers to them. Even incrementally updating all pointers encountered during query processing to their new memory slot does not allow to reclaim the old memory slot as there may still be other pointers to them. As we cannot allow these slots to accumulate infinitely, wasting memory space and fragmenting object storage, we have to deal with them regularly. We use the compaction thread to deal with them in the background after the compaction is finished and the reclamation epoch ended. As direct pointers are only used between self-managed objects, the compaction thread can update all direct pointers by enumerating over each self-managed collection and for each reference in the object, check if the pointer is pointing to a memory slot that has the forwarding flag set and, if this is the case, update the pointer to the new memory slot. As compaction is usually performed one collection at a time and, for each self-managed type, it is known at compile time what self-managed types reference it, it is not necessary to scan all self-managed collections. Instead, it is sufficient to only scan the ones that store types that reference the type of the collection that has been compacted. Furthermore, only the references to that type have to be checked and updated. As all of this information is known at compile time, we automatically generate compaction functions for each type in a C♯-to-C♯ compiler. After the relocation phase, the compaction thread then executes the functions generated for the type stored in the collection that has been compacted on all self-managed collections that contain direct-

pointers that could have been affected by the compaction. The compaction thread does not have to suspend any application processing while updating direct pointers in a collection.

Until they are dealt with, we have to prevent old object slots from becoming potential reclamation candidates. When an object is relocated, instead of turning its object slot into a regular limbo slot, as was the case for the compaction strategy presented in Section 5.5, the memory slot is turned into a limbo slot with the maximum removal timestamp. The latter is reserved only for this purpose and not reachable otherwise and does not count towards the memory block's reclamation threshold. Once the compaction thread has updated all pointers, the removal timestamp is set to the correct removal epoch. In this case, it is two epochs after the relocation epoch as this was the last epoch that threads could have accessed the object's data.

6.5 Columnar storage

In Section 4.5, we have shown that, for black-box collections, replacing row-wise storage with a columnar layout can improve query evaluation performance. Columnar storage in black-box collections was enabled by decoupling the object-oriented type representing the data stored in the collection with the layout the data was actually stored in the collection's data store and making the code generator aware of the collections internal data layout. The same is also possible for self-managed collections; however, it requires the following two changes to the system:

- As manually managed objects in columnar self-managed collections are stored in a vertically decomposed layout, the JIT compiler has to be aware of this layout and inject the corresponding code to access columnar data.
- The code generator has to be aware of the columnar data layout and also has to generate code that iterates over the columnar data and follows references in a columnar-aware manner.

In order to require as little change as possible to the manual memory management system and to self-managed collections, we do not use traditional columnar storage for self-managed collections, but instead employ PAX. Employing PAX allows us to identify every self-managed object by the identifier of its memory block and its memory slot in that block. Both identifier are identical to the ones in a row-wise data layout, but each memory block is divided into several consecutive memory areas that each

represent a column and allow access to the object's fields by casting the memory area to an array and then using the objects slot identifier to index this array, as was the case for the slot directory and back-pointer before. To adapt the memory manager and self-managed collections for columnar storage, we have to replace the pointer to the object's memory slot that is stored in the indirection table with the object's block and slot identifier. This is necessary because there is no single pointer to the object's data anymore, but one for each field (column). To access the data of an object, we look up its memory block using an array of memory blocks indexed by their block identifier, and then use the slot identifier to find the position of the value in its column. The memory areas that store columns are reachable by a constant offset in the data section of each memory block.

6.6 Evaluation

We now evaluate a prototype of self-managed collections (SMCs) implemented in C$^\sharp$. Our prototype is not integrated into the managed runtime as proposed in Section 6.1, but instead implemented as a library using `unsafe` C$^\sharp$ code. As we did not change the JIT-compiler to automatically inject the code for correctly dereferencing references to self-managed objects, we have to add this code by hand when accessing self-managed objects outside the generated query code. The generated code automatically adds these checks in `unsafe` C$^\sharp$ code.

As before, our benchmarks are based on the TPC-H benchmark. Self-managed collections store data elements as manually-managed objects which could use a reference-based or join-based data representation. However, as the support for references are the major difference between self-managed and black-box collection and the reference-based representation embodies typical object-oriented design, we only test self-managed collections using this representation.

So far, we only compared our approaches against C$^\sharp$'s `List<T>` collection type. From our experience, `List<T>` is the best performing collection type that ships with C$^\sharp$. However, this comparison is no longer fair when it comes to self-managed collections. Self-managed collections are inherently thread-safe when adding or removing objects. This is not the case for `List<T>` or most of the default collection types that ship with C$^\sharp$. For these collection types, it is up to the programmer to ensure that the collection is used in a thread-safe manner when utilizing multiple application threads. For this reason, we also compare self-managed collections to some of the thread-safe collection

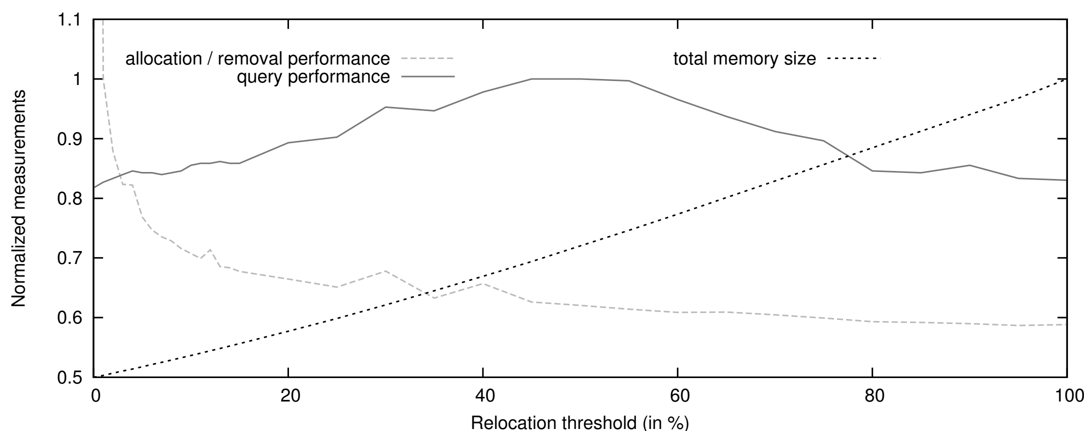


Figure 6.3: Performance characteristics of self-managed collections when varying a memory block's relocation threshold. Measurements normalized to their maximum value with the exception of the allocation / removal performance which is normalized to a 1% relocation threshold to improve readability

types that ship with C♯. However, the choice of thread-safe collection types in C♯ is limited and only `ConcurrentBag<T>` and `ConcurrentDictionary<TKey, TValue>` provide comparable functionality. However, the former does not allow the removal of specific objects.

As some of the experiments presented here highlight the weaknesses of storing database-like collections in a memory space managed by automatic garbage collection, we first have a look at the configuration options available. .NET supports two garbage collection modes: *workstation* and *server*. Both modes support either interactive (concurrent) or batch (non-concurrent) garbage collections. In our tests the server modes consistently outperformed the workstation ones, so we only report results for the server mode and only report both concurrency settings if their results differ.

6.6.1 Sensitivity to relocation threshold

In Section 5.4, we introduced the *relocation threshold*. It specifies the percentage of memory slots in a data block that may be occupied by limbo slots before the system adds the block to the reclamation queue to initiate their reclamation. The actual reclamation is then performed by the allocation function once at least two epochs have passed to ensure that at least as many limbo slots as specified by the threshold can be safely reclaimed. Varying this threshold affects the memory size, the cost of memory operations and the query performance of self-managed collections. In Figure 6.3, we

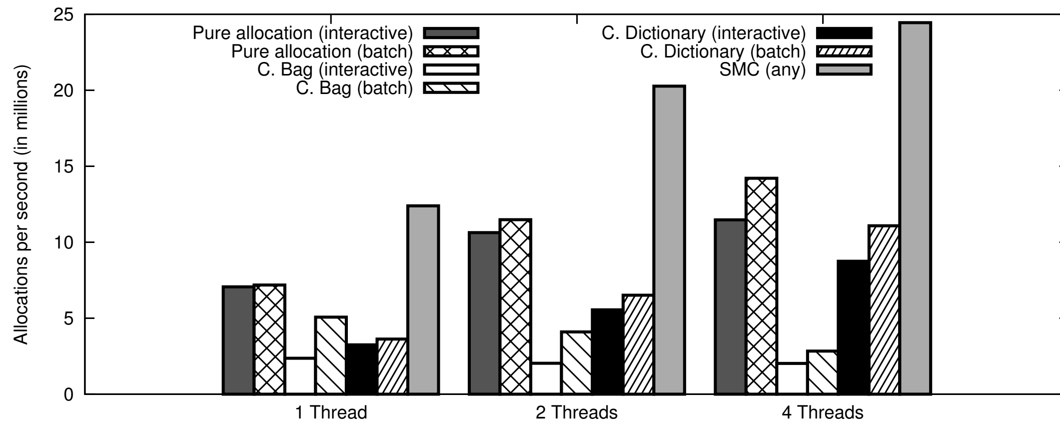


Figure 6.4: Batch allocation throughput

show how these factors change when varying the threshold. The measurements are normalized to their maximum values with the exception of the allocation / removal performance which is normalized to a 1% relocation threshold to improve readability. As the percentage of unused limbo slots grows, so does the memory footprint of the collection. The cost of performing memory operations (i.e., insertions and removals) slowly decreases with an increasing threshold as allocations have to scan less memory slots to find a slot that can be reclaimed. Query performance seems to be less dependent on the additional slot directory entries that have to be processed with an increasing threshold, but more on the branch misprediction penalties when verifying if the slot is occupied. At a 50% threshold, the branch predictor has the most trouble to correctly predict if the slot is occupied. Based on the results of Figure 6.3, we will use a 5% threshold for the following experiments. For a 5% threshold, the memory requirements of self-managed collections are comparable to that of storing managed objects in `List<T>`.

6.6.2 Evaluating collection primitives

In Figure 6.4, we compare the throughput (in objects per second) of allocating `lineitem` objects (using the default constructor) in an SMC to the pure allocation throughput of managed objects in .NET[1] and the throughput of allocating managed objects and adding them to a concurrent collection. For managed allocations we report the throughput for interactive and batch garbage collection; the latter consistently provides better performance. SMCs perform significantly better than both managed collections and even

[1] We use pre-allocated, thread-local arrays to prevent objects from being garbage collected.

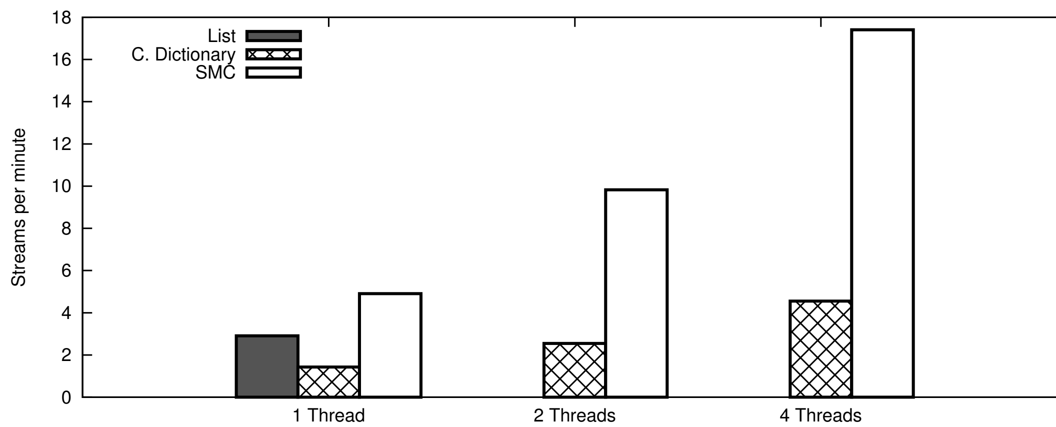


Figure 6.5: Combined throughput of both refresh streams

better than the pure allocation cost of managed objects. All objects remain reachable so the runtime performs numerous garbage collections, with many of them stopping all other application threads, to copy objects from younger to older generations. SMCs allocate from (previously unused) thread-local blocks, which reduces the synchronization overhead of multiple allocation threads to about one atomic operation per 10k `lineitem` allocations.

To measure the throughput of memory operations we use an adapted version of TPC-H's refresh streams. Each thread continuously runs one of two kinds of streams with the same frequency. The first stream type creates and adds `lineitem` objects (0.1% of the initial population) to the `lineitem` collection. The second stream type enumerates all elements in the `lineitem` collection and removes 0.1% of the initial population based on a predicate on the object's `orderkey` value. All 0.1% objects to delete are provided in a hash map and removed in a single enumeration over the collection. This benchmark represents a common use case of refreshing the data stored in self-managed collections. In Figure 6.5, we report the stream throughput (in streams per minute) for SMCs against `ConcurrentDictionary<TKey, TValue>`; `ConcurrentBag<T>` is not included because it does not support the removal of specific elements. SMCs perform better than both types of managed collections in all cases.

Out of the two garbage collection settings reported in Figure 6.4, the (non-concurrent) batch mode provides the higher throughput. In other garbage collection intensive benchmarks, we found the batch mode to enable a several times higher throughput. However, the higher throughput comes at a price: response time. Where concurrent collectors (interactive) can perform big parts of the garbage collection on a background thread without pausing all application threads, non-concurrent collectors have to pause

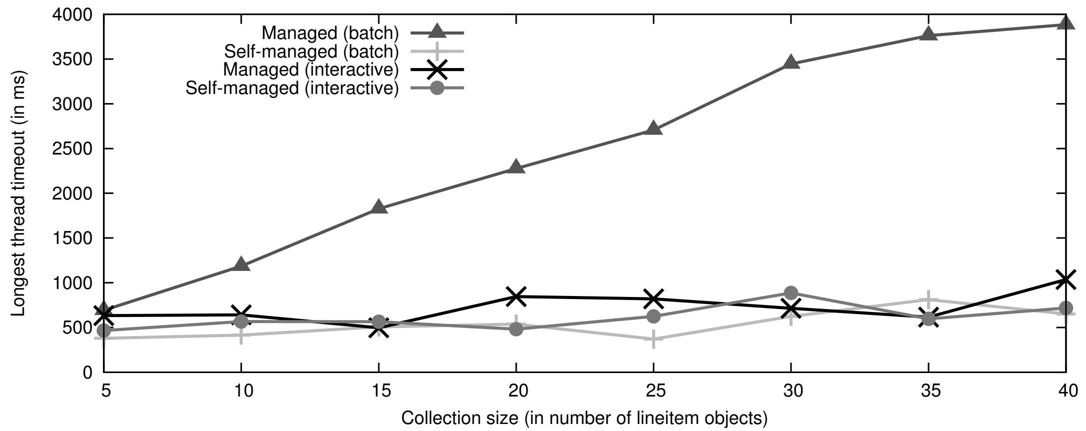


Figure 6.6: Timeouts caused by garbage collection

all threads for the entire duration of the collection. As the size of the managed heap grows, so does the duration of full garbage collections and, hence, the application's maximum response time. To illustrate this, we insert a number of objects into a collection, either managed or self-managed, and then start two threads in parallel. The first thread continuously allocates managed objects with varying lifetimes and the second continuously sleeps for one millisecond and measures the time that passed in the meantime. If it observes that significantly more time has passed than expected, it records the value as it most likely was caused by garbage collection triggered by the other thread. We show the maximum timeout measured for a varying number of objects stored in the collection in Figure 6.6. For non-concurrent garbage collection, the maximum timeout increases with a growing number of objects stored in a managed collection, but remains fairly stable when these objects are stored in a SMC instead. It shows that the duration of garbage collections increases with growing data volumes stored in the managed heap. In the batch mode this negatively impacts the responsiveness of the application; in the interactive mode, it negatively impacts the overall application performance as the background collection thread steals processing resources from the application. In both cases, using SMCs improves the scaling when facing an increasing data volume.

6.6.3 Enumeration and query performance

We first report the pure enumeration performance of self-managed collections before considering more complex queries. Our queries either: (*a*) enumerate the `lineitem` collection and perform a simple function on each object to ensure that all `lineitem`

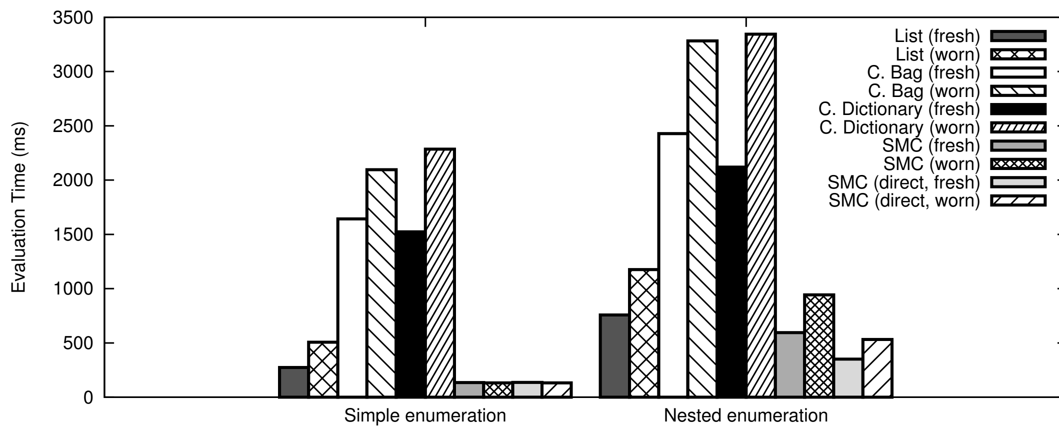


Figure 6.7: Enumeration performance

objects are accessed (simple enumeration); or (*b*) enumerate the `lineitem` collection, and for each object follow the `order` reference to a `customer` object and perform a simple function on the latter to ensure that `customer` objects are also accessed (nested enumeration). Query performance deteriorates over time as objects are added and removed from the collection. In managed collections, objects may end up scattered all over the managed heap, whereas in SMCs the blocks containing objects may have holes due to limbo slots. In Figure 6.7 we show the performance of both query types after the collections are freshly loaded (fresh) and after the collections have undergone numerous (typically three times the number of objects in the collection) removals and insertions of new objects (worn). Each removal and insertion step removes a random object from the collection, creates a new object that is a copy of the removed object and then adds the newly created object to the collection. SMCs (indirect) outperform all automatically managed collections. However, when performing nested object accesses, the difference with `List<T>` diminishes because of the additional memory access required by the indirection step when following self-managed references. By utilizing the direct pointers of Section 6.4, we can bypass this look-up and improve performance. When comparing the fresh and worn states, SMCs only lose performance under nested accesses, whereas managed collections exhibit reduced enumeration performance in both cases. As `ConcurrentDictionary<TKey, TValue>` is the best performing thread-safe managed collection, we exclude `ConcurrentBag<T>` in what follows.

In Figure 6.8, we show the performance of the first six TPC-H queries. For managed collections, we report the query performance of compiled C$^\sharp$ code on the reference-based data representation. We report on two versions of compiled code for SMCs:

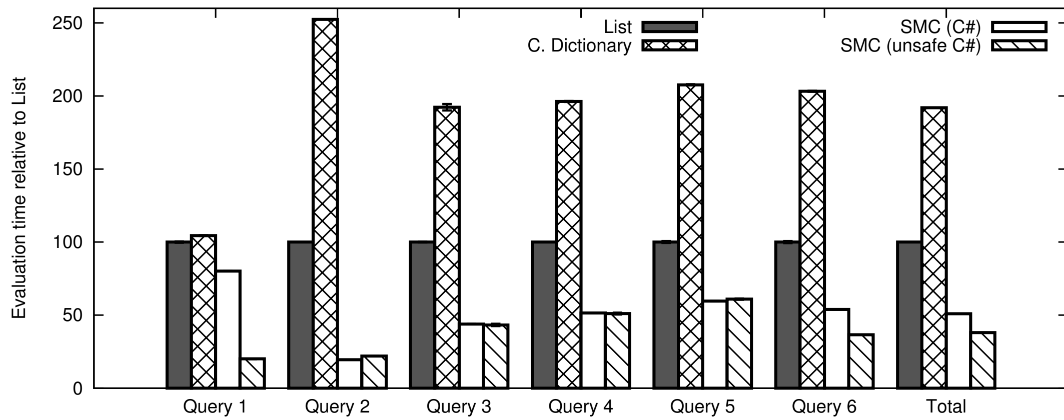


Figure 6.8: TPC-H Queries 1 to 6

(*a*) Compiled C♯ code that, other than the enumeration code, is equivalent to the code used for managed collections. This illustrates the fraction of the overall improvement contributed by the better enumeration performance of SMCs. (*b*) Compiled `unsafe` C♯ code that contains optimizations only possible on SMCs. One such optimization is to use direct pointers to primitive types in an object (e.g., `decimal` values) as arguments to functions that operate on them (e.g., addition). For managed objects, these functions have to be called by value as the garbage collector may move the object inside the managed heap at any time without notice and, hence, the pointer would become invalid. Another optimization is to use memory regions [Gay and Aiken, 1998] for all intermediate data during query processing, which improves performance by excluding those intermediates from garbage collection. In Figure 6.8, we report the query processing performance relative to the performance of `List<T>`. SMCs perform significantly better than `ConcurrentDictionary<TKey, TValue>`, the fastest competing thread-safe collection in .NET; and even between 47% and 80% better than `List<T>`. For all queries, storing data in self-managed collections rather than as objects in the managed heap significantly improves query performance, even if only compiling (safe) C♯ code. Similarly to what has been the case for black-box collections, this performance benefit is mostly caused by the more cache and prefetching efficient data layout. Enumerations on the primary collection(s) access objects in the order in which they are stored in memory and successively accessed objects are (mostly) stored in consecutive memory locations. However, this is not the case when following references to objects stored in secondary collections. In this case memory accesses are randomly, but in most real-world applications (as is the case for the TPC-H data set), there is still some locality when following references. On top of the performance benefits caused by the data

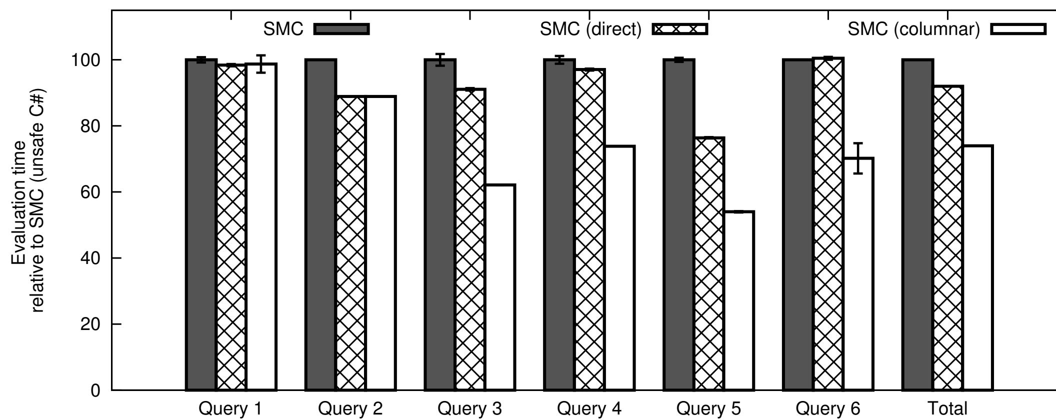


Figure 6.9: Direct pointer and columnar storage

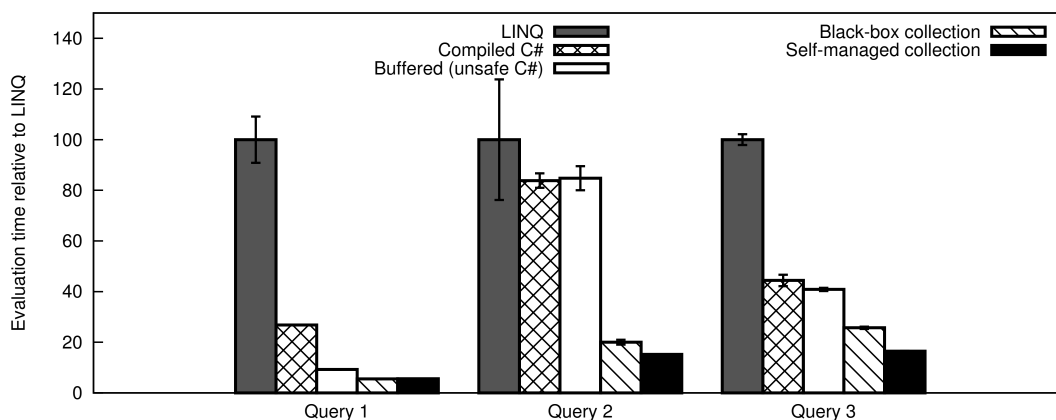


Figure 6.10: Query performance of TPC-H queries 1 to 3

layout of self-managed collections, query 1 again benefits greatly from having direct pointer access to the data objects when performing `decimal` computations.

In Figure 6.9, we show the impact of the direct pointer optimization introduced in Section 6.4 and columnar storage as discussed in Section 6.5. Direct pointer moderately improve query performance for queries that contain joins, in particular for query 5, by reducing the additional random memory access that has to be payed when accessing related objects through references. Columnar storage (also using direct pointers) shows further improvements similar to what has been the case for black-box collections by better utilizing the memory read from main memory into the CPU. Note that, in contrast to the columnar implementation for black-box collections, we rather use a PAX-like [Ailamaki et al., 2001] layout to represent columnar data than a pure columnar layout. PAX layouts do not vertically partition the entire table, but instead only vertically partition the data elements within the same memory block.

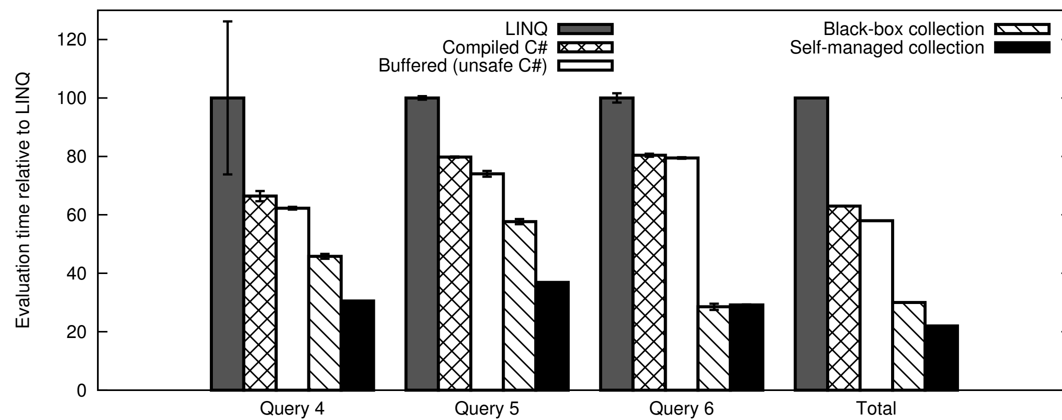


Figure 6.11: Query performance of TPC-H queries 4 to 6

In Figures 6.10 and 6.11, we compare the direct pointer version of the same six queries with the best performing approaches presented in previous chapters. Other than for black-box collections which inherently rely on a join-based data representation, all reported results rely on a referenced-based representation. Of the approaches presented in this thesis, self-managed collections provide the best querying performance. They feature the advantages of black-box collections, but as all data elements are objects, they allow accessing related elements through references which improves the query performance of join-heavy queries.

6.7 Related work

6.7.1 Object-oriented databases

As self-managed objects store object-oriented data in a database-inspired storage layer, they share some common characteristics with object-oriented databases (OODBMSs). OODBMSs gained traction in the 80s and early 90s. They were intended to address the needs of an emerging class of non-traditional application areas [Carey et al., 1986]. These areas include computer aided technologies (e.g., CAD or CAM), multimedia applications or artificial intelligence systems. The main requirements posed by these emerging applications were persistence of object-oriented data with transactional consistency guarantees and support for accessing older versions of object-oriented data. [Zand et al., 1995] compare features and implementation details of the major commercial and research OODBMSs that existed at the time. In the following, we will have a closer look at some of these object-oriented database systems to illustrate the similari-

ties and differences between them and self-managed collections.

Object-oriented data cannot be represented on disk in the same way as it is in memory because references between nested objects in memory are implemented by pointer to virtual memory addresses whereas the same is not possible for disk addresses. OODBMSs typically maintain unique object identifier (OID) for all persistent objects. References to nested objects are implemented by storing the referenced object's OID. Some systems use logical surrogates for OIDs, e.g., [Kim et al., 1990] uses the pair `<class identifier, instance identifier>`, others use physical surrogates, e.g., [Carey et al., 1986] use the pair `<disk page number, slot number>`. Because of this difference in memory and disk representation of object-oriented data, many OODBMSs manage objects in two different formats, a disk format and an in-memory format, e.g., [Kim et al., 1990]. Objects are fetched from persistent storage based on their OID. Most systems using logical surrogates maintain some form of table that contains information on what objects are in the buffer pool. Objects that are not in the buffer pool are fetched from physical storage. Some systems prevent reusing OIDs to handle dangling references, e.g., [Hornick and Zdonik, 1987].

The ORION database system [Kim et al., 1990] divides the buffer pool of the database into a page buffer and an object buffer. The page buffer is managed at the disk page level and stores objects in the disk format. The object buffer stores objects in their in-memory format. When the application accesses an object, the storage manager is responsible to bring the page that contains the object into memory and copy it into the in-memory format of the object buffer. The application directly interacts with objects in the object buffer pool through calls using an object's unique OID. The transaction manager feature ensures consistency of accesses to these objects. Instances of the same class are clustered in physical storage. Queries are defined against a single target class, but also allows nested access to referenced objects of other classes.

The ODE object database [Agrawal and Gehani, 1989] is built on the database programming language O++, which is based on C++. O++ extends C++ by adding the option to create persistent objects in a similar manner as volatile C++ objects are created. Instead of pointers, persistent objects are referenced using their unique identifier. Pointers to persistent objects can only refer to other persistent objects and pointer to volatile objects can only refer to other volatile objects. Objects of the same type are clustered together in physical storage and subclusters can be manually defined to further refine the clustering. Queries are processed in `for` loops over clusters or subclusters that provide integrated query syntax to evaluate predicates and to define an

order.

EXODUS [Carey et al., 1986] provides software tools to facilitate the semi-automatic generation of application-specific database systems. The storage layer handles data elements unaware of their object-oriented representation. Storage objects are uninterpreted variable-length byte sequences. Database consistency is provided at this level. The custom database code of the system is written in the database programming language E, which is based on C. E provides support for persistent objects and allows the database engineer to handle them like regular objects. The E compiler is responsible to add the relevant code to bring the required data of the persistent objects into memory buffers and ensure database consistency. The E compiler translates E code into C code. The storage manager improves query performance by accepting performance-related hints, e.g., to place a new objects in close proximity to a specific existing object. File objects, which are collections of persistent objects, are used to group objects together. This allows queries to iterate over them and the storage manager to cluster them in physical storage by placing all objects in disk pages allocated to a file. Files may only contain objects of a single class, however, this also includes inherited classes. ENCORE/ObServer [Hornick and Zdonik, 1987] also uses a typeless back-end that is responsible for managing the persistent object store.

The object-oriented database systems described in this section share some common characteristics with self-managed collections. The main similarity is that both systems handle a duality of storage types. In OODBMSs, the duality refers to volatile and persistent objects whereas for self-managed collections it refers to automatic and self-managed objects. OODBMSs that provide deep integration with the application such as ODE [Agrawal and Gehani, 1989] also use a compiler to automatically address the duality by adding the required code. On top of this, the indirection table exhibits similarities to the table used in OODBMSs to map logical OIDs to physical storage. Finally, OODBMSs reduce the cost of disk accesses by clustering related data together. However, self-managed collections are built for a different purpose. Instead of providing persistence, transaction consistency and object versioning in the presence of multiple clients, they are designed to improve query processing performance inside a single application.

Chapter 7

Conclusion and discussion

The previous chapters have presented several strategies to process data in the memory space of a managed application. We started off by having a look at the execution model of LINQ-to-objects and recognized that it exhibits similar inefficiencies to the volcano iterator model used in traditional relational database systems. To address these inefficiencies, we leveraged query compilation techniques similar to those that have been proposed in the database community to eliminate these inefficiencies as discussed in Chapter 3. Our experiments showed improvements in query evaluation performance compared to LINQ-to-objects of up to a factor of three. However, we also identified additional inefficiencies that could be addressed by processing queries using low-level techniques. Since garbage collection managed objects do not allow us to use these techniques, we introduced a staging phase in the compiled query code to make query-relevant managed data accessible to optimized low-level query code. Evaluating this approach showed a performance improvement in query evaluation of up to an order of magnitude compared to LINQ-to-objects.

Motivated by our findings when evaluating the staging approach, in Chapter 4, we identified further inefficiencies that are caused by automatic garbage collection and chose to address them by replacing automatic memory management for the data elements in a collection with the in-memory storage layer of a relational database system. As a result, we introduced black-box collections. Black-box collections hide away the relational data store by utilizing existing object-relational mapping techniques and improve query evaluation performance by allowing queries to directly operate on the data in the data store. However, as data is stored in a relational fashion, black-box collections do not allow references and, hence, require implicit join operations to access related data elements together. Despite this disadvantage, our experiments showed an

improved query evaluation performance compared to all previous approaches. Compared to the staging approach, which so far provided the best performance, query evaluation performance improved by up to a factor of four. We also compared the query evaluation performance of black-box collections to that of a modern commercial database system and achieved superior query performance for five of the six tested queries.

Black-box collections showed the best query evaluation performance of all approaches covered so far, however, they also expose additional overhead to the application developer as they have to deal with the object-relational mapping and the restrictions that come with it. To address this and to allow the developer to deal with application data as objects that can be referenced in the application in a similar way to managed objects, we introduced self-managed collections. Self-managed collections exhibit different semantics to regular managed collections in order to allow them to manually manage allocations and deallocations of all contained objects using a type-safe manual memory management system. The semantics of black-box collections couple the lifetime of all objects stored in the collection with their containment in the collection. We introduced the safe manual memory management system in Chapter 5 and self-managed collections in Chapter 6. We evaluated self-managed collections together with some basic operations of the manual memory management system and saw that self-managed collections exhibit superior performance. Furthermore, due to allowing references between related objects, self-managed collections also outperformed black-box collections by up to 50%.

This thesis has shown that there is huge potential for improving the evaluation performance of LINQ queries on data stored in the memory space of a managed application. The strategies that we presented to achieve this showed a varying degree of performance improvements and required a varying degree of programmer awareness and involvement. A good approach should provide excellent query performance, but also has to be as transparent as possible and pose as few restrictions as possible to the programmer. Out of our approaches, we believe that self-managed collections provide this balance as they allow the necessary steps to improve query performance, but also manage data as objects and, hence, pose very little restriction on the programmer. However, generating pure C♯ code, as presented in Section 3.4, requires the least programmer involvement as the approach is entirely built on existing C♯ technologies and hides away the query compilation using the query provider facility.

There are other factors that impact query evaluation performance that we did not

look at in this thesis. Some of them were assumed, but not explicitly dealt with. This includes, for example, query optimization based on heuristics and statistics on the data stored in a collection. As expression trees are comparable to query trees in database systems and the input collections of a query are accessible in the query provider through the expression tree, we assume that query optimization can be added to queries on collections in the same way as in relational database systems. However, instead of implementing query optimization, we merely assumed its existence and started all our experiments using a hand-optimized LINQ query plan based on LINQ's method syntax. The optimized LINQ statement is based on optimized query plans from commercial database systems and manual measurements. Future work could provide a full implementation of a query optimizer inside a LINQ query provider and a collection type that gathers collection statistics and provides the query optimizer with access to them. Statistics could either be collected through an explicit `GatherStatistics` call or by incrementally updating them when objects are added and removed. The implementation could further add support to automatically cache and reuse intermediate and final results, similar to [Nagel et al., 2013].

Another option to improve query performance that has not been covered is parallel query evaluation. In all our experiments, all queries are evaluated using a single thread. However, as LINQ queries (using a query provider) are declarative and, hence, only specify the query result and not how to obtain it, the code generator can employ existing parallelization techniques from the database space (e.g., [Dees and Sanders, 2013]) to evaluate queries using multiple threads. This improves query performance by better utilizing modern processors that typically support four or more hardware threads. We believe that multi-threaded query evaluation strategies from database systems can be directly applied to query compilation of LINQ queries. One interesting aspect here is reference-based joins. The performance benefits when parallelizing query evaluation are limited by query operations that require synchronization between different threads that perform the same task. For example, parallelizing a hash-based join operation requires all threads to insert data elements in the same hash table or to pre-partition each thread's key range to later build an individual hash table for each key partition. Reference-based joins, on the other hand, do not require any synchronization to perform the joins and, therefore, exhibits a greater degree of parallelism.

Other options to improve the evaluation performance of LINQ queries are to introduce indexes or physical clustering of data elements based on a sort key. The former can be added as a feature of the collection type. The query optimizer then uses the ex-

istence of indexes and the collection's statistics to decide what access path to use when evaluating a query. Physical clustering is not possible when storing data as managed objects. Having the storage layer of a database system, black-box collections can support clustering. The same is also possible for self-managed collections, but because clustering requires under-full blocks to be merged and over-full ones to be split, the required relocation epochs to allow the movement of objects between these blocks are likely to cause a significant overhead.

Some applications might also require more database features than assumed here. These include persistent storage of objects or transactional consistency guarantees. Future iterations of our approaches could integrate these features. Another interesting area of future research is on how to achieve these characteristics using emerging non-volatile memory technologies, as suggested in [Viglas et al., 2014].

Bibliography

[Abadi et al., 2009] Abadi, D. J., Boncz, P. A., and Harizopoulos, S. (2009). Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665.

[Agrawal and Gehani, 1989] Agrawal, R. and Gehani, N. H. (1989). Ode (object database and environment): the language and the data model. In *ACM SIGMOD Record*, volume 18, pages 36–45. ACM.

[Ailamaki et al., 2001] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. (2001). Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180.

[Ailamaki et al., 1999] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Wood, D. A. (1999). DBMSs on a modern processor: Where does time go? In *VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number DIAS-CONF-1999-001, pages 266–277.

[Austin et al., 1994] Austin, T. M., Breach, S. E., and Sohi, G. S. (1994). *Efficient detection of all pointer and array access errors*, volume 29. ACM.

[Boncz, 2002] Boncz, P. A. (2002). *Monet; a next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam.

[Boncz et al., 2005] Boncz, P. A., Zukowski, M., and Nes, N. (2005). MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237.

[Boyapati et al., 2003] Boyapati, C., Salcianu, A., Beebee Jr, W., and Rinard, M. (2003). Ownership types for safe region-based memory management in real-time Java. *ACM SIGPLAN Notices*, 38(5):324–337.

[Braginsky et al., 2013] Braginsky, A., Kogan, A., and Petrank, E. (2013). Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 33–42. ACM.

[Braginsky and Petrank, 2011] Braginsky, A. and Petrank, E. (2011). Locality-conscious lock-free linked lists. In *Distributed Computing and Networking*, pages 107–118. Springer.

[Carey et al., 1986] Carey, M. J., DeWitt, D. J., Frank, D., Muralikrishna, M., Graefe, G., Richardson, J. E., and Shekita, E. J. (1986). The architecture of the EXODUS extensible DBMS. In *Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 52–65. IEEE Computer Society Press.

[Chamberlin et al., 1981] Chamberlin, D. D., Astrahan, M. M., King, W. F., Lorie, R. A., Mehl, J. W., Price, T. G., Schkolnick, M., Griffiths Selinger, P., Slutz, D. R., Wade, B. W., et al. (1981). Support for repetitive transactions and ad hoc queries in System R. *ACM Transactions on Database Systems (TODS)*, 6(1):70–94.

[Cheney et al., 2013] Cheney, J., Lindley, S., and Wadler, P. (2013). A practical theory of language-integrated query. In *ACM SIGPLAN Notices*, volume 48, pages 403–416. ACM.

[Copeland and Khoshafian, 1985] Copeland, G. P. and Khoshafian, S. N. (1985). A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM.

[Dees and Sanders, 2013] Dees, J. and Sanders, P. (2013). Efficient many-core query execution in main memory column-stores. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 350–361. IEEE.

[DeLine and Fähndrich, 2001] DeLine, R. and Fähndrich, M. (2001). Enforcing high-level protocols in low-level software. *ACM SIGPLAN Notices*, 36(5):59–69.

[Desnoyers et al., 2012] Desnoyers, M., McKenney, P. E., Stern, A. S., Dagenais, M. R., and Walpole, J. (2012). User-level implementations of read-copy update. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):375–382.

[Dhurjati et al., 2003] Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. (2003). Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Notices*, 38(7):69–80.

[Diaconu et al., 2013] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. (2013). Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM.

[Dragojević et al., 2014] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. (2014). FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI*, volume 14.

[Francisco et al., 2011] Francisco, P. et al. (2011). The Netezza data appliance architecture: a platform for high performance data warehousing and analytics. *IBM Redbooks*.

[Fraser, 2004] Fraser, K. (2004). *Practical lock-freedom*. PhD thesis, University of Cambridge.

[Freedman et al., 2014] Freedman, C., Ismert, E., and Larson, P.-Å. (2014). Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Eng. Bull.*, 37(1):22–30.

[Freytag and Goodman, 1989] Freytag, J. C. and Goodman, N. (1989). On the translation of relational queries into iterative programs. *ACM Transactions on Database Systems (TODS)*, 14(1):1–27.

[Gay and Aiken, 1998] Gay, D. and Aiken, A. (1998). *Memory management with explicit regions*, volume 33. ACM.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.

[Graefe, 1994] Graefe, G. (1994). Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120–135.

[Grossman et al., 2002] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. (2002). Region-based memory management in Cyclone. In *ACM Sigplan Notices*, volume 37, pages 282–293. ACM.

[Grust et al., 2010] Grust, T., Rittinger, J., and Schreiber, T. (2010). Avalanche-safe LINQ compilation. *Proceedings of the VLDB Endowment*, 3(1-2):162–172.

[Gupta et al., 2015] Gupta, A., Agarwal, D., Tan, D., Kulesza, J., Pathak, R., Stefani, S., and Srinivasan, V. (2015). Amazon Redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1917–1923. ACM.

[Herlihy et al., 2005] Herlihy, M., Luchangco, V., Martin, P., and Moir, M. (2005). Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196.

[Hornick and Zdonik, 1987] Hornick, M. F. and Zdonik, S. B. (1987). A shared, segmented memory system for an object-oriented database. *ACM Transactions on Information Systems (TOIS)*, 5(1):70–95.

[Ivanova et al., 2010] Ivanova, M. G., Kersten, M. L., Nes, N. J., and Gonçalves, R. A. (2010). An architecture for recycling intermediates in a column-store. *ACM Transactions on Database Systems (TODS)*, 35(4):24.

[Kim et al., 1990] Kim, W., Garza, J. F., Ballou, N., and Woelk, D. (1990). Architecture of the ORION next-generation database system. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):109–124.

[Klonatos et al., 2014] Klonatos, I., Koch, C., Rompf, T., and Chafi, H. (2014). Building efficient query engines in a high-level language. In *Proceedings of the VLDB Endowment*, volume 7.

[Krikellas et al., 2010] Krikellas, K., Viglas, S. D., and Cintra, M. (2010). Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613–624. IEEE.

[Lieberman and Hewitt, 1983] Lieberman, H. and Hewitt, C. (1983). A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429.

[Manegold et al., 2000] Manegold, S., Boncz, P. A., and Kersten, M. L. (2000). Optimizing database architecture for the new bottleneck: memory access. *The VLDB JournalThe International Journal on Very Large Data Bases*, 9(3):231–246.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195.

[Meijer et al., 2006] Meijer, E., Beckman, B., and Bierman, G. (2006). LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM.

[Michael, 2004] Michael, M. M. (2004). Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504.

[Murray et al., 2011] Murray, D. G., Isard, M., and Yu, Y. (2011). Steno: automatic optimization of declarative queries. In *ACM SIGPLAN Notices*, volume 46, pages 121–131. ACM.

[Nagel et al., 2014] Nagel, F., Bierman, G., and Viglas, S. D. (2014). Code generation for efficient query processing in managed runtimes. *Proceedings of the VLDB Endowment*, 7(12):1095–1106.

[Nagel et al., 2013] Nagel, F., Boncz, P., and Viglas, S. D. (2013). Recycling in pipelined query evaluation. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 338–349. IEEE.

[Neumann, 2011] Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550.

[Padmanabhan et al., 2001] Padmanabhan, S., Malkemus, T., Jhingran, A., and Agarwal, R. (2001). Block oriented processing of relational database operations in modern computer architectures. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 567–574. IEEE.

[Pirk et al., 2013] Pirk, H., Funke, F., Grund, M., Neumann, T., Leser, U., Manegold, S., Kemper, A., and Kersten, M. (2013). CPU and cache efficient management of memory-resident databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 14–25. IEEE.

[Rao et al., 2006] Rao, J., Pirahesh, H., Mohan, C., and Lohman, G. (2006). Compiled query execution engine using JVM. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 23–23. IEEE.

[Sompolski et al., 2011] Sompolski, J., Zukowski, M., and Boncz, P. (2011). Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40. ACM.

[Tofte and Talpin, 1997] Tofte, M. and Talpin, J.-P. (1997). Region-based memory management. *Information and computation*, 132(2):109–176.

[Ungar, 1984] Ungar, D. (1984). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Sigplan Notices*, volume 19, pages 157–167. ACM.

[Viglas et al., 2014] Viglas, S., Bierman, G. M., and Nagel, F. (2014). Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Eng. Bull.*, 37(1):12–21.

[Viglas, 2014] Viglas, S. D. (2014). A comparative study of implementation techniques for query processing in multicore systems. *Knowledge and Data Engineering, IEEE Transactions on*, 26(1):3–15.

[Wadler, 1988] Wadler, P. (1988). Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer.

[Wadler, 1990] Wadler, P. (1990). Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer.

[Wanderman-Milne and Li, 2014] Wanderman-Milne, S. and Li, N. (2014). Runtime code generation in Cloudera Impala. *IEEE Data Eng. Bull.*, 37(1):31–37.

[Zand et al., 1995] Zand, M., Collins, V., and Caviness, D. (1995). A survey of current object-oriented databases. *ACM SIGMIS Database*, 26(1):14–29.

[Żukowski, 2009] Żukowski, M. (2009). *Balancing vectorized query execution with bandwidth-optimized storage*. PhD thesis, University of Amsterdam.