Using Relational Algebra to Facilitate Safe Refactoring for Performance in Rust

Eric Michael Sumner

Faculty of Industrial Engineering, Mechanical Engineering, and Computer Science
University of Iceland
2021

Using Relational Algebra to Facilitate Safe Refactoring for Performance in Rust

Eric Michael Sumner

60 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Computer Science

MS Committee
Snorri Agnarsson
Hjálmtýr Hafsteinsson

Master's Examiner
Björn Þór Jónsson

Faculty of Industrial Engineering, Mechanical Engineering, and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, June 2021

Using Relational Algebra to Facilitate Safe Refactoring for Performance in Rust
Relational Algebra in Rust
60 ECTS thesis submitted in partial fulfillment of a *Magister Scientiarum* degree in Computer Science

Faculty of Industrial Engineering, Mechanical Engineering, and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Tæknigarður, Dunhaga 5
107 Reykjavík
Iceland

Telephone: 525 4000

Bibliographic information:
Eric Michael Sumner, 2021, *Using Relational Algebra to Facilitate Safe Refactoring for Performance in Rust*, Master's thesis, Faculty of Industrial Engineering, Mechanical Engineering, and Computer Science, University of Iceland, pp. 149.

Printing: 1
Reykjavik, Iceland, May 2021

Abstract

The choice of how to organize data in memory has a significant effect on the overall efficiency of computer programs. Changing requirements over time may require this choice to be revisited after a program's initial development has been completed. Historically, data retrieval routines and program logic have been strongly coupled, which hinders the maintenance programmer's ability to reorganize a program's data without introducing new logic faults.

This paper proposes a new framework, Memquery, which separates these two concerns, allowing a program's internal data to be reorganized without major changes to its logic routines. The theoretical basis for this framework is relational algebra, which has served a similar role in database systems for the past 50 years. A prototype of the Memquery framework is presented. A comparison study with traditional development techniques demonstrates that this prototype is capable of producing more maintainable programs with similar performance characteristics.

Útdráttur

Skipan gagna í minni tölvunnar hefur veruleg áhrif á keyrsluhraða forrita. Kröfur um virkni sem breytast í tímans rás geta valdið því að breyta þarf skipan gagna eftir að frumgerð er forrituð. Venjan er sú að gagnameðhöndlun sé sterklega samofin annarri virkni hvers forrits, sem gerir forritara sem vinnur í viðhaldi forritsins erfiðara að breyta skipan gagna án þess að valda því að nýjar villur læðist inn.

Í þessari ritgerð er kynnt nýtt kerfi, Memquery, sem aðskilur þessi tvö vandamál og gerir kleift að endurskipuleggja innri gögn forrits án verulegra breytinga á annarri virkni. Fræðilegi grunnurinn fyrir þessu kerfi er venslaalgebra, sem hefur þjónað samsvarandi hlutverki í gagnagrunnskerfum síðastliðin 50 ár. Frumgerð af Memquery kerfinu er kynnt. Samanburður við hefðbundnar þróunaraðferðir sýnir að þessi frumgerð gerir kleift að þróa forrit sem eru auðveldari í viðhaldi en hafa sambærilegan hraða.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This thesis would not have been possible without the support I received from many people along the way. I would like to thank the members of my committee for their efforts in reviewing this work. My supervisor, Snorri Agnarsson, was instrumental in helping to refine my original ideas into a coherent whole. He also graciously provided the Icelandic translations of the thesis's title and abstract. Additionally, I would like to thank my parents for their continuous emotional support during this process and the members of the *Rust Programming Language Forum*[1], who were always there to help when I encountered technical problems.

1 https://users.rust-lang.org

1 Introduction

One common aspect across engineering projects of all disciplines is that maintenance costs tend to dominate the costs of initial development; software development is no exception. 50 years ago, David Parnas recognized this and introduced the concept of information hiding as a mechanism to improve the maintainability of software [1]. Around the same time, E. F. Codd published his formulation of relational algebra to improve the maintainability of database systems [2]. Over the intervening half-century, both of these papers have come to be viewed as seminal works in their respective domains and the concepts they introduced are still widely relevant to modern practice. Despite this, few people have noted that these two papers fundamentally address the same problem: Isolating the semantic logic of a program from the particular storage strategy used for the required data.

Their two solutions differ in character, however: Parnas develops a set of guidelines for API design that can facilitate the ability to replace one storage implementation with another, and relies on the judgment of the programmer to keep the guidelines in mind when designing an API for a new domain. Codd, on the other hand, is considering only a single class of program: database storage systems. He develops a mathematical model for these databases that allows the database users to have an expressive, semantic interface to the stored data while the database administrators remain free to change the underlying structure to support changing workloads.

Modern algebraic type systems are powerful enough to model Codd's relational algebra directly [3], and this can be used to facilitate Parnas' information hiding mechanically. This provides similar advantages to software maintenance as the adoption of relational databases has brought to database administration. In particular, as the capabilities and usage patterns of a piece of software evolve, the original data storage and access strategy often becomes suboptimal. The use of relational algebra in the original design allows this strategy to be updated with minimal coding effort, which reduces maintenance costs in two ways:

- Coupling between program logic and data access routines within a program is reduced, which reduces the likelihood that a data model change will introduce logic faults.
- Data access algorithms can be written once and deployed as modular components in many programs. Any maintenance effort applied to these reusable modules then benefits many programs instead of just one.

1.1 Organization of This Document

This paper describes Memquery, a prototype library intended to demonstrate that the relational model is a viable alternative to traditional approaches for managing a program's in-memory data. The remainder of this chapter discusses prior work in this area.

Memquery is implemented in Rust. As a relatively new programming language, Rust may be unfamiliar to many readers; chapter 2 provides an informal overview of Rust's core features.

Relational algebra demands more complicated relationships between types than is necessary for typical Rust programs. A domain-specific language, Tylisp, has been developed to provide a more ergonomic syntax for expressing these complicated relationships between types. Chapter 3 describes the usage and features of Tylisp.

Chapter 4 describes the Memquery library itself. Each subchapter consists of a topic overview followed by reference-style descriptions of each defined type and trait.

To demonstrate the efficacy of Memquery , chapter 5 presents a comparison study between programs implemented with Memquery and Rust's standard-library collections.

Chapter 6 discusses the limitations of the current Memquery library, and what work would be necessary to further develop it into a production-ready, rather than prototype, library.

1.2 Information Hiding

In his paper, Parnas discusses several different software design problems: A system for producing keyword-in-context (KWIC) indices, a Markov algorithm compiler, and a Markov algorithm interpreter. For each of these problems, he describes two candidate modularizations.

The first modularization represents common practice circa 1971. It includes one module for each processing step, and the output of each module serves as the input to the next module in the sequence. Because data stored in core memory serves as the interface between modules, its format must be precisely defined before implementation work can begin on the individual modules.

The second modularization demonstrates *information hiding*, Parnas' proposed criteria for dividing a program into modules. Instead of steps in a process, Parnas envisions each module as a service which encapsulates one "difficult design decision." The interface for each module is designed to be independent of the particular choice made by the module's implementor.

In many cases, this approach naturally produces a hierarchy: Modules at any given level depend only on modules in lower levels of the hierarchy, and the lowest-level modules operate independently. This allows for a subtree of modules to be easily transplanted from one program to another.

Parnas demonstrates this with the Markov compiler and interpreter: In the traditional modularizations, there is little common structure that could be shared between the two implementations. When viewed as a set of services, however, there are several modules that appear in both programs, such as a mechanism to store rules and a pattern matcher that can interpret those rules.

Information hiding also allows design decisions to be revisited later in development. For the KWIC indexer, Parnas identified a number of questionable design assumptions and examined which modules would be affected if they were to change. In each case, the necessary changes were confined to a single module in the information-hiding modularization, but spanned many modules in the traditional modularization.

For the purposes of this paper, it is interesting to note that these potential design changes primarily concern the internal memory layout of the program. For example:

- Should all text lines be loaded into memory, or should they be read from disk on demand?
- Should characters be stored individually, or should they be packed several to a machine word?
- Should circular shifts be realized, or should they be stored as an index into the line storage?

1.3 Relational Algebra as Information Hiding

Codd's paper is concerned not with program structure, but with the challenges of interacting with databases. Nevertheless, Codd comes to the same conclusion as Parnas: Because the internal representation of data will inevitably change over time, an abstract interface to the data must be provided which allows access to many different internal representations. From his abstract:

> "Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). ... Most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.
>
> *-- from "A relational model of data for large shared data banks." [2]*

These statements hold true whether the "large data banks" are stored on disk or in core memory.

Codd goes on to describe three kinds of data dependencies that relational algebra is intended to break: ordering, indexing, and access path dependence. In modern programs, each of these dependency types remains a common source of coupling between data storage and application logic routines.

Ordering dependence occurs when the program logic is dependent on the particular order records appear in the data store. If it becomes necessary to alter this stored ordering, the application code will break. In many programs, this is mitigated by declaring that records are stored in an arbitrary, unspecified order; algorithms that rely on a particular ordering must first sort the records according to the algorithm's needs. If a useful storage ordering is later adopted, however, these extra sorting steps need to be found and removed manually.

Indexing dependence occurs when the program logic is aware of the particular indices that are present on a data store. If an index is added, algorithms that could benefit from using the index must be located and modified to gain any benefit. If an index is removed, any algorithm that relies on that index must be rewritten to either use a different index or a sequential scan. The difficulty of maintaining consistency between an index and the main data store often leads programmers to use unindexed data structures, even when an index would be beneficial.

Access path dependence occurs when program logic needs to know one piece of data in order to access some other, related piece of data. This most often occurs when data is stored in a mapping, such as a hashtable: In order to access a property, the

corresponding hash key must be known. In cases where the hash key is not known, the program must explicitly search all entries in the table. If the mapping key is changed, all routines that use the map must be updated to use the new key.

Relational algebra is designed to present a uniform interface for accessing conceptually tabular data, which can be efficiently implemented for a wide variety of internal representations. Each table, known as a relation, is characterized by a set of named columns and an unordered set of records. In addition to a name, each column specifies the domain of values that it may contain; every record in a relation contains one value for each named column of its relation.

The overall goal of a relational-algebra based system is to provide a property that Codd calls *symmetric exploitation*: The operations required to obtain a given subset of stored data should not rely on the internal representation used by the database. Instead, a program should only need to know which relations exist and the columns that any given relation contains.

To this end, relational algebra defines a number of operators that can be used to refine the set of relations stored in a database into a single relation that contains only the data

relevant to a single subprogram, such as:

- *Projection*, which involves striking out or reordering columns.
- *Selection*, which involves striking out records which do not meet a given condition.
- *Joining* two relations, which involves finding pairs of records where the columns common to both relations contain equal values.

The particular implementation of these operators will be necessarily dependent on the internal representation used by the database, but those details can be hidden from database users. This is exactly the property that Parnas' concept of *information hiding* requires for defining a module's interface.

1.4 Prior Art: Language-Integrated Query (LINQ)

Memquery is not the first attempt to bring relational-algebra concepts to general purpose programming. Designed by Erik Meijer and integrated into Microsoft's .NET suite of languages, LINQ promises to provide a unified interface to all kinds of data sources [4]. From a user's perspective, this is certainly the case: Meijer relaxed some of Codd's constraints to produce a more general system based on relational algebra; a system that, when used within an appropriate domain, produces identical results. These generalizations, however, cause problems for both users and implementors of new data sources.

1.4.1 Symmetric Exploitation

One key design principle of relational algebra is the ability for every field in the datastore to be used in any logical position. Codd termed this symmetric exploitation.

The ability for every field in the datastore to be used in any logical position within a query. Meijer, for instance, describes how to write a LINQ provider that only supports a single, extremely specific, form of query [5]. The ability to transpose the two where clauses is "left as an exercise for the reader". This omission wasn't just for sake of brevity: Oren Eini provides a case study of writing several real-world LINQ providers; its conclusion recommends that system designers "choose a conventional way to define each supported operation. [They] will be much better off if ... users' input can be made to follow specific patterns." [6]

Part of the justification for this approach is that the API shouldn't pretend to be capable of queries that it can't execute performantly. On the contrary, symmetric exploitation is key to maintaining a separation of concerns between the capabilities of the data store and the operational logic of the program. When the query structure is required to match the internal layout of the data provider, it is impossible to change the data layout without also changing the query. Instead of expressing semantic intent directly, these queries describe a particular strategy for fetching data. The maintenance programmer must then infer the original intent of each query from this strategy and accompanying documentation. This increases the cost and risk of adjusting the internal data layout, which is often a necessary component of efforts to improve performance.

1.4.2 Practical Deficiencies

LINQ's design also causes practial problems when trying to implement data providers. An implementor must choose between two interfaces, `IEnumerable` or `IQueryable`. Often, neither is ideal.

LINQ's ability to query `IEnumerable` types is the core of its ability to provide a unified query interface. `IEnumerable`'s simplicity, however, prevents LINQ from making optimized queries. `IEnumerable` is .NET's standard iteration interface, implemented by many core types ranging from generic collections to the file system interface. To be applicable to so many disparate types, `IEnumerable` is a lowest-common-denominator interface: It has a single method that returns a cursor. This cursor can do only three things: retrieve the current item, step to the next item, and rewind to the beginning. If a query needs to skip over some items, there is no faster way than one at a time. Neither is there any facility for ordering, which would be required to halt a query early.

To provide a more sophisticated query implementation, a data provider must instead implement the `IQueryable` interface. This interface produces a cursor from a provided a syntax tree. The language described by this tree is quite complex. There are no less than 25 different node types, including lambda definitions, operators, goto statements, and method calls [8]. Each of these node types has its own set of properties, possible child nodes, and variants. Eini describes the process of implementing `IQueryable` as

equivalent to writing a compiler. Even worse, .NET provides no facililty to execute subexpressions; every provider must implement the whole language itself.

In practice, then, the prospective implementor is faced with a dilemma: He/she can implement `IEnumerable` easily and provide full functionality to users, but with fundamentally limited performance. Or, they can implement `IQueryable`, which will require a large quantity of code with many branching paths, code that, due to its inherent complexity, will undoubtedly be the source of many bugs.

1.4.3 Memquery's Solution

Memquery addresses these problems by using a separable intermediate representation for queries. Each query is represented as an intersection of filter clauses. Custom data sources need only provide a specialized implementation for those clauses they can efficiently implement. All the other clauses are applied in a final filtering step, without the data source needing to know what test is being applied. For example, one clause may be used for a local BTree index lookup while another is handed to the provider stored in the BTree value and yet a third is used to filter the ultimate results, all within the same query (cf. §4.6).

Instead of a syntax tree, Memquery gains its flexibility from a series of lazy adapter objects. Each of these objects represents a relational operation that has been applied to one or more source relations. It is a fully-functional view, or derived relation, that operates lazily: When it receives a query request, it dispatches some related query to its sources, which may in turn be other views. It then applies a transformation to each row returned from these queries to produce its own result rows (cf. §4.7).

2 The Rust Programming Language

Rust is a relative newcomer to the world of systems programming languages. Originally developed as a research project by Mozilla in 2010 to address the shortcomings of C and C++, it is now an independent project supported by several industry partners including Amazon, Huawei, Google, and Microsoft [9].

2.1 Design Principles

Rust's design is guided by a desire to make high-performance software with a minimal maintenance burden. Many of these design decisions are based the observation that several kinds of defect require a disproportionate amount of effort to correct. Rust, therefore, makes an effort to systematically minimize the occurrence of these particularly troublesome defects [10].

These defects fall into two broad categories: inconsistent behavior and forward-compatibility hazards. Defects that exhibit inconsistent behavior may appear to work correctly some or most of the time, but occasionally cause the program to fail. Forward-compatibility hazards are not defects *per se*, but have the potential to become defects due to changes elsewhere that appear innocuous.

The mechanisms that Rust has put in place to combat these defects usually impose a compile-time cost only. For times when even this minimal cost is too great, however, Rust provides an escape hatch: By using the `unsafe` keyword, a programmer can exempt sections of code from some of these extra requirements. This keyword, then, signals the presence of code that needs to be manually checked for correctness.

2.1.1 Inconsistent Behavior

Inconsistent behavior can occur whenever a program interacts with a complex or opaque system, such as a memory allocator, thread scheduler, or program optimizer. These systems generally provide narrowly-drafted guarantees about their behavior. Presented with a situation outside these guarantees, the system will often still behave as the programmer might expect. Under these conditions, however, small external changes may cause a program to produce incorrect results without warning.

Accessing memory after it has been formally returned to the allocator is known as a use-after-free fault. If the allocator has not yet reissued the memory for another allocation, there will be no apparent error. In Rust, as in most programming languages, the particular strategy used by the allocator is not specified and subject to change for any reason. A particular sequence of user inputs or compiling a program for a different

architecture, for example, may cause a previously innocuous use-after-free to manifest as an error. This error will not necessarily occur anywhere near the problematic code: Any allocation in the program can become corrupted due to an erroneous write somewhere else. Rust uses a system of ownership and lifetime annotations (§2.2.3) to ensure that no outstanding references to an object exist at the point when its memory is returned to the allocator for potential re-use.

Improper synchronization between multiple threads of execution can lead to a data race, where one thread writes to memory concurrently with a read in another thread. If the two concurrent accesses happen to occur at different times, the program will appear to operate correctly. Sometimes, though, the thread scheduler may interrupt the writing thread while it is in the middle of modifying the shared memory, which then allows the reading thread to see an inconsistent state. To guard against this kind of problem, Rust tags every type with whether or not it is synchronized for concurrent access, and references to unsynchronized memory are prevented from being transmitted between threads.

There are too many kinds of inconsistent behavior to enumerate them here. These two examples are representative of Rust's strategy for preventing this kind of programming error: Information relevant to deciding whether or not an action should be allowed is encoded in the type system, and problematic operations are only allowed to operate on values of an appropriate type.

2.1.2 Forward Compatibility

Another kind of long-reaching defect stems from the independent development of interoperating modules. If an application programmer relies on some property of a software library, then the library author can no longer alter that property without breaking the application code. This problem is particularly acute when the library is widely distributed and used by many different applications. To combat this, Rust treats type signatures as explicit contracts between library and application programmers. Every type constraint serves a dual purpose: it is a minimum requirement for code that wishes to call the function and a listing of all the capabilities the function implementation is allowed to rely upon.

Polymorphism in Rust is based on traits, which are similar to interfaces in other languages. Each trait represents some capability that a type has, and type bounds are expressed as a series of required traits. There is deliberately no mechanism for requiring the absence of a trait, which gives library programmers the freedom to add trait implementations without risk of breaking application code. At the interface between modular parts, Rust generally requires explicit annotations for any property that affects the situations where that code can be used, even if it theoretically could be determined automatically. This gives library authors the ability to reserve a portion of the design space for future changes, without fear of breaking users' code. Perhaps a concrete example will make this more clear:

If a type implements the `Copy` trait, then the compiler knows that the value can be safely copied using a bit-for-bit copy with no additional procedures. Some examples of `Copy` types are integers and shared references. Certain other types fundamentally can't ever be `Copy`: Reference-counted pointers (`Rc`) need to update the count whenever they're duplicated and exclusive references (`&mut`) can't be duplicated at all without violating non-aliasing guarantees.

Compound types could theoretically implement `Copy` automatically if all of their fields also implement `Copy`. If this were done, however, it would create a software evolution hazard: A type could implement `Copy` by accident because its implementation doesn't happen to include any non-`Copy` fields. The developer is then constrained from changing the implementation to add a non-`Copy` field without potentially breaking downstream users who were relying on the `Copy` functionality. Rust instead allows developers full latitude for changing their implementations and provides tools to opt into restrictions on themselves that allow downstream users more flexibility.

2.1.3 Integrating Manual Verification

No automated analysis mechanism, however, is capable of verifying all correct programs. Moreover, these non-analyzable programs often have desirable properties, such as a fast execution speed. Rust sidesteps this problem by allowing the programmer to mark sections of code for human verification: the unsafe keyword. Code within these unsafe sections are allowed to do certain things that are disallowed outside them.

> "The only things that are different in Unsafe Rust are that you can:
>
> - Dereference raw pointers
> - Call `unsafe` functions (including C functions, compiler intrinsics, and the raw allocator)
> - Implement `unsafe` traits
> - Mutate statics
> - Access fields of unions
>
> -- from *The Rustonomicon*, §1.2 "What Unsafe Rust Can Do" [10]

There are primarily two ways to attach semantic meaning to a data type. The first is to express it via the type's name and documentation. From the compiler's perspective, this is a completely implicit definition: If the type is misused, there is no way for the compiler to notice. The alternative, which is preferred Rust style, is to restrict which code is allowed to construct each type. If there are only a few different constructors, then semantic information can be encoded in the assertions those constructors make: If a value of the type exists, it must be valid.

In order for user-specified types to maintain useful invariants, the language must provide a way for the type's author to restrict the changes that can be made. Rust

accomplishes this via a fine-grained privacy system. By default, access to individual types, fields, and functions is restricted to the module that defines them. Additional modules can be provided access to these items by annotating them as public. This allows library authors to maintain the integrity of type invariants by disallowing access to the type's underlying fields and instead providing access through constructors and methods that enforce the relevant invariants.

Due to the expressiveness of Rust's type system, these invariants can often be expressed as type constraints, which allows the compiler to reject programs that violate invariants instead of producing a runtime error. This allows for more performant final code by omitting runtime validity checks: If a value of a type exists, then its invariants have been upheld. For example, Rust's `String` type is internally a vector of bytes. All safe methods to construct a `String`, however, ensure that this vector contains valid UTF-8 encoded data. Any functions that operate on `Strings`, then, can safely ignore the possibility that they will encounter data of some other encoding.

If a function is marked as safe to call, it may still contain unsafe code: This is an assertion by the programmer that all necessary preconditions are enforced mechanically. Much of Rust's standard library, for instance, relies on unsafe code internally but presents a safe API for application programmers. Jung, et al. formally verified the soundness of this approach in general, and also confirmed that the standard library's implementations are correct [11].

2.2 Data Model

When reasoning about Rust code, it is often helpful to think in terms of *typeclasses*, sets of types that all share some common properties. Rust has several different mechanisms to describe typeclasses, depending on the nature of the properties that they share:

- The class of types with a particular set of fields is represented by a *generic datatype*.
- The class of types which exhibit a particular behavior is represented by a *trait*.
- The class of types which are valid for a particular period of time is represented by a *lifetime annotation*.

In generic code, an externally-specified type is represented by a type parameter. This parameter is a local identifier that acts in all ways like a concrete type name. The types that are allowed to bind to a particular parameter are specified by a special kind of expression, a type bound. Each type bound specifies one or more typeclasses; any type that is a member of all the listed typeclasses is valid.

Within the scope of a type parameter, it may only be used in a way that will be valid for all possible type bindings. This analysis is completely local: The compiler rejects code that attempts to use any property that is not provided by one of the typeclasses in corresponding bound expression.

2.2.1 Compound Datatypes

Arrays `[`*T*`;`*N*`]` and slices `[`*T*`]` represent a contiguous region of memory that contains several instances of a uniform type *T*. Arrays contain a fixed number of elements *N*, which must be known at compile time. Slices, on the other hand, may contain any number of elements. Because they have a size known only at runtime, slices are subject to certain restrictions in where they may be used; these restrictions are discussed in more detail in §2.2.5.

Tuples `(`*A*`,`*B*`,...)` are *ad-hoc* compound types which represent a sequence of values with dissimilar types *A, B, etc*. Tuple types are automatically defined by their argument types, and do not need to be explicitly declared before use.

Structures (`struct`) are explicitly-named types with a common memory layout. Each structure definition specifies zero or more fields, which can be either named or unnamed.

Enumerations (`enum`) are explicitly-named tagged union types. At any given moment, exactly one of the variants is present. Each variant has a unique name and can contain zero or more fields, just like a `struct`. Field access is only possible through pattern-matching operations which ensure that the appropriate variant is present.

Both structures and enumerations can be defined generically. The compiler monomorphizes these definitions: Each unique combination of type parameters used in the program produces a different type with its own memory layout.

Closures are anonymous types that implement one or more of the function-call traits `Fn`, `FnMut`, and `FnOnce`. Each closure expression produces a distinct type which contains all of its captured variables. By default, these variables are captured by reference, which prevents the closure object from leaving the activation record it was defined in (cf. §2.2.4). The `move` keyword changes this behavior to capture-by-value, which instead transfers ownership of the captured variables to the closure object. Because these types are anonymous, they must be bound to a generic parameter in order to be stored or passed from one function to another.

2.2.2 Traits

Generic programming in Rust relies on traits, which describe an interface that can be implemented by multiple types. Traits themselves can be generic, by taking type or lifetime parameters, and can be restricted to being implemented only by types that otherwise meet certain conditions, described by type bounds. For most purposes, each possible binding of parameters to a generic trait definition is treated as an independent

trait. In particular, it is never possible to elide the parameters of a trait: they must always be explicitly specified.

Traits are also often used to represent a typeclass with properties the compiler is unable to enforce. When analyzing code that refers to one of these traits, different situations call for different assumptions about the trait's implementation. In a safety analysis, the general practice is to assume that the implementation may contain any non-`unsafe` code that the compiler would accept. When evaluating program logic, however, the assumption is that the trait implementation is correct, i.e. that it satisfies all of the defining properties of the corresponding typeclass.

Each trait defines a number of *associated items* that describe the shared interface. Most of these items are function or method prototypes that describe the operations that a type must support in order to implement the trait. These items can also be secondary type definitions or constant values. Each of these items can be subject to arbitrary bounds by the trait definition, but they can otherwise be freely chosen by any of the trait's implementors.

Trait type parameters and associated types often get confused, and it can sometimes be difficult to tell which is most appropriate for any given situation. In essence, a type parameter is conceptually chosen by the caller and an associated type is conceptually chosen by the implementation. By specifying something as an associated type, you are asserting that it is uniquely determined by the combination of the trait, including its generic parameters, and `Self`, the type that implements the trait.

The usual role of an associated type in a trait is to serve as the return type of a method: Each trait implementation is free to choose a different return type for the method, as long as it satisfies the bounds in the trait definition. Code that uses this method on a generic type is then only allowed to perform operations that can be inferred by these bounds, unless it further constrains the associated type.

There is a subtle difference between specifying a type bound on an associated type and specifying it on the method that returns that type. Consider the following traits:

```
trait T1 {
    type Output: Condition;
    fn do_something(&self)->Self::Output;
}

trait T2 {
    type Output;
    fn do_something(&self)->Self::Output
        where Self::Output: Condition;
}
```

`T1` requires that every implementation of the trait must specify a `Output` type that implements the `Condition` trait. `T2`, on the other hand, places no constraints on

`Output`. Instead, it specifies that `do_something` is only a valid operation if `Output` implements the `Condition` trait.

When reasoning about traits, it is important to realize that the Rust compiler always treats the types that implement a trait as an open set: It assumes that there are always some unknown types that also implement the trait. These types could be added to the program at any time, and might not share any undeclared properties that the known implementations may have in common. Thus, the compiler will not accept code that relies on such undeclared commonalities.

2.2.3 Ownership and Lifetimes

Rust's memory model is based on a system of ownership semantics. "Ownership" in Rust is the right to either destroy an object or to relocate it to another memory location. With only a few exceptions, every object is owned either by a function's activation record or by some other object which is transitively owned by an activation record. Because each value has exactly one owner, it is structurally impossible to destroy any object multiple times, which would lead to a double-free error. Also, because essentially every object is transitively owned by an activation record, memory leaks are extremely rare.

Often, it is cumbersome or incorrect to give a function the rights attached to object ownership. If an object is stored inside a vector, for example, it would need to first be removed from the vector, which is a potentially expensive operation. To accommodate this and similar cases, an activation record is allowed to provide temporary access to any object that it owns, a process known as "generating a borrow." The compiler produces an internal identifier, a *lifetime*, which represents the region of the function body in which the borrowed object remains valid.

Generating a borrow then produces one of two primitive types that are annotated with a special kind of generic parameter, a *lifetime annotation*: a shared reference (`&'a` *T*) or an exclusive reference (`&'a mut` *T*). Lifetime annotations are denoted by a single quote preceding the parameter name, which is customarily in lower case (`'a`). It is important to understand that the seemingly concrete type `&'a str`, for example, is really a generic type with a free parameter `'a` and therefore represents a typeclass. The value of this parameter is some concrete lifetime, attached to a particular activation record.

Any compound type that contains one of these reference types must specify its lifetime parameter explicitly. There is only one non-generic lifetime in Rust: `'static` indicates the borrowed value will remain valid until the program exits. All other lifetimes come from the borrow-generation process described above and cannot be named: Each invocation of a function conceptually creates a new, distinct, set of lifetimes. The compound type, then, must take a lifetime annotation of its own as a generic parameter in order to fill the reference's lifetime parameter.

A special phase of the compiler, the *borrow checker*, rejects any program where a value of an annotated type might exist outside its corresponding region of validity. Once this phase is complete, all remaining references in the program have been proven correct and there is no more need of the lifetime annotations. They are then discarded, and have no further effect. In particular, neither memory layout nor code generation is affected by the lifetime system.

This model does, however, impose some constraints on algorithm design. Some algorithms, especially those which involve arbitrary graph structures, must be modified in ways that could impact performance. The overall cost of these changes is usually a small constant factor, with no change in the algorithm's asymptotic behavior.

2.2.4 Reference Types

Like references in garbage-collected languages, Rust references always point to a valid target. The two systems differ in how they achieve this goal: Garbage-collected languages use a runtime system to delay the destruction of and object until some time after the last reference to that object expires. Rust, on the other hand, lets an object's owner fully control when it is moved or destroyed; the compiler uses static analysis to ensure no reference to the object exists at that point.

The two reference types, `&'a` *T* and `&'a mut` *T*, differ in how the compiler computes the extent of the lifetime `'a`. An exclusive reference's region ends as soon as the borrowed object is used in any way that does not pass through the reference. A shared reference's region, on the other hand, only ends when the borrowed object is moved or destroyed, or an exclusive borrow is generated for the same object.

As mutating operations generally require an exclusive reference, this forms a kind of statically-enforced read-write lock: Code that holds an exclusive reference can safely assume that no intermediate states will be observed by outside code; code that holds a shared reference can safely assume that the object will not change while the reference is held.

Some algorithms are too complicated to be fully analyzed by these static mechanisms. Rust's standard library provides some facilities to support these more complicated algorithms with runtime support. A brief discussion of these APIs will both elucidate how the lifetime system works in practice and provide an introduction to some of Rust's common utilities.

The shared ownership types, `Rc<`*T*`>` and `Arc<`*T*`>`, behave more like references in garbage-collected languages, and are used where there isn't a clear hierarchical ownership structure. They are reference-counted owning pointers that will only destroy the contained *T* once the last reference to it is destroyed. The only difference between them is that `Arc` is thread-safe by virtue of using atomic operations on its internal reference count, which may impose a performance penalty.

An Rc<*T*> (or Arc<*T*>) can be readily used to obtain a shared reference to its contained *T*, thanks to its Deref::deref method, which has a signature of fn<'a>(&'a Rc<T>)->&'a T.[2] This states that the output reference to *T* is valid within the same region 'a as the input reference to the Rc<*T*>. This intuitively makes sense: As long as at least one copy of the Rc<*T*> object exists, its internal reference count will remain above zero and the contained *T* will not be destroyed.

Because this pattern of projecting a reference with one lifetime into another reference of the same lifetime is so common, Rust lets you omit the annotations in this case: deref's signature is more commonly written as fn(&Rc<T>)->&T, which is entirely equivalent to the form given above.

Obtaining an exclusive reference to the contained *T* is not so straightforward. The Rc<T>::get_mut method has a signature of fn(&mut Rc<T>)->Option<&mut T>. It returns an Option because creating an exclusive reference might not be possible: There may be other Rc's pointing to the same allocation. If the internal count is one, however, this is the only copy of the Rc<*T*>; it's safe to produce the exclusive reference.

To ensure that the returned reference will retain exclusive access, calling deref or any similar method on the Rc<*T*> must be blocked. This is accomplished by requiring an exclusive reference as the input parameter: Though the exclusive reference &'a mut Rc<*T*> no longer exists after get_mut returns, the return type is annotated with the lifetime 'a.[3] This informs the compiler that it should prevent the returned value from existing outside the region 'a. Because there are no other accesses to the original Rc<*T*> within 'a, there is no possibility of concurrent access to its contained value.

2.2.5 Dynamic Dispatch and Runtime Type Introspection

By default, Rust uses static dispatch for accessing a trait's methods and other related items. This requires the compiler to know the concrete type of the object being used, which is not always possible. For traits that meet certain requirements, Rust defines a special type known as a *trait object*, which can represent any concrete type that implements the trait. For a trait *T*, this is written as "dyn *T*".

A trait object has an identical memory layout as the concrete type that it is abstracting. As these various concrete types may have different sizes, trait objects are *dynamically-sized*. This severely restricts how they can be used. In particular, a dynamically-sized type (DST) can only exist behind a layer of indirection, such as a reference.[4] Internally, pointers to DSTs contain a second field that holds additional information about the target type. In the case of trait objects, this second field contains the address of the relevant vtable. By storing the vtable address inside the pointer, Rust avoids the need

2 T is a free parameter here, because the Deref trait is implemented for Rc<T>.

3 More specifically, it would appear when written in a fully-explicit form.

4 There is one exception: A DST may also be stored directly as the last field of a struct, in which case the struct itself is dynamically-sized and must be behind a layer of indirection.

to store this address inside the concrete type's layout: The cost of dynamic dispatch is borne only by code that explicitly uses the feature.

In most cases, it is impossible to recover the concrete type that backs a trait object. This introspection is only possible via the `Any` trait. The trait object dyn Any has two methods that attempt to convert references back into a concrete type, `downcast_ref<T>` and `downcast_mut<T>`. These return `Option<&T>` and `Option<&mut T>`, respectively. If the backing type of the `dyn Any` object is `T`, they return `Some(...)`, otherwise they return `None`.

Due to a limitation in the implementation of `Any`, it is only implemented for types that conform to a `'static` bound. Every type in a Rust program is assigned a `TypeId` at compile time. Any compares T's id to its backing type's id and allows the conversion only if they are equal. As soon as the borrow checker has verified that the program does not violate any lifetime constraints, the lifetimes are discarded; the `TypeId`s are generated after this stage of compilation has finished. As such, two types that differ only in their lifetime annotations will receive the same id. In order to maintain memory soundness, `Any` needs to be implemented for only one concrete type for each id. The `'static` bound achieves this by requiring all of a type's embedded lifetime annotations to be `'static`.

Note that this doesn't require the backing objects themselves to last forever. This is a common misconception among newcomers to Rust. Instead, the `'static` bound prevents the object from referring to some external value stored on the stack. A lifetime bound on a type *T* only says that all values of type *T* must be destroyed within the named region.

2.3 Compile-Time Programming

Rust has several mechanisms that work together at compile time to enable high-level abstractions to produce simplified machine code. These include macros for syntax manipulation, bounds for expressing relationships between types, constant expressions, and traditional program optimization.

Macros can expand to arbitrary code but operate purely at a syntax level. Each macro invocation must be entirely self-contained: There is no mechanism to transfer information between multiple macro calls. This leaves the compiler free to calculate macro results in any order. Macro invocations can only occur at defined positions within the Rust grammar, and must always expand to syntax that is valid in that position. Rust has two ways to specify macros, *macros by example* and *procedural macros*.

Macros by example are defined via a template language which can be embedded into any Rust source file. Each definition specifies one or more syntax templates and corresponding replacements. Every macro variable in a template is annotated with a

production rule from the main Rust grammar and may only appear in the macro expansion in a position where that production is valid.

In contrast, procedural macros are both more powerful and more cumbersome to write. Procedural macros are written as ordinary Rust functions which consume and produce syntax tree objects. They must be defined in a separate library, which is then dynamically linked into the Rust compiler itself.

Type bounds form a Turing-complete language for describing arbitrary relationships between types (cf. Appendix A). This is particularly useful for keeping track of statically-known properties of values as they are transferred between different code modules. `String` is a good example of this: It maintains an invariant that its contents are always UTF-8 encoded. Functions that interact with `String`s can then trust that they have UTF-8 encoded data without performing their own validation, regardless of any other code that the object may have passed through since the `String` object was originally created (cf. §2.1.3).

A significant subset of Rust can be evaluated in a `const` context, including loops, conditionals, and function calls. These `const` expressions are evaluated at compile time. In addition to being precomputed, these can also be used in certain places where a runtime expression is forbidden, such as the length of statically-allocated arrays. This system is still under active development; Rust 1.51[5], for example, adds a limited ability for constants to be used as generic parameters, alongside types and lifetimes.

Finally, Rust relies heavily on traditional compiler optimization strategies. Generic functions and structure definitions are monomorphized, and a separate implementation is emitted for each set of type parameters that appears in the program. Any constant value derived from these parameters will be known to the optimizer. If these are used in a conditional expression, the optimizer can remove the unreachable branch. The non-aliasing rules for references can also be exploited by the optimizer: Multiple reads from the same reference will always produce the same result, for example; it can be cached in a register instead of performing a second read from memory.

5 Released March 2021

3 Tylisp: A Type-System Embedded Programming Language

The arguments to relational operators have more complicated preconditions than those of typical Rust functions. In particular, relations and relational tuples are primarily characterized by their relational header, which is a set of columns. Relational operators may impose conditions on the union or intersection of these sets. It is imperative, then, to be able to model these more complicated relationships within Rust's type system.

As Rust's type system is Turing complete (see Appendix A), it can be used to describe any computable relationship between two types. Its syntax, however, is optimized to express simple relationships between types. During the development of Memquery, it quickly became obvious that a more ergonomic means of expressing complicated type relationships was required, a domain-specific language.

Tylisp was developed to meet this need. It allows Rust types to be treated as ordinary values, and every Tylisp expression evaluates to a Rust type. Lisp's homoiconicity makes it a natural choice for the design: If Tylisp expressions are also Rust types, then the evaluation engine can be written natively inside Rust's type system.

3.1 Heterogeneous Lists

The primary data structure in any Lisp-derived language is the heterogeneous list [12], and Tylisp is no exception. These are composed of *cons cells*, which have two fields of differing types, and the empty list. Unlike its counterparts in other Lisp-derived languages, `HCons` directly contains both the head and tail of the list. This requires the compiler to be able to statically determine the types of all the list's constituent items. The compiler will lay out the entire list as a single, contiguous block of memory without any pointer indirections.

Tylisp defines a number of facilities to work with these lists in ordinary Rust code. The `sexpr!` and `sexpr_val!` macros provide a shorthand for constructing list types and values, the `List` and `Take` traits allow manipulation of individual elements within a list, and the `ListOf` and `ListOfRefs` traits produce iterators over a list's elements.

3.1.1 sexpr! Macro

The `sexpr!` macro expands to a Rust type composed of a tree of cons cells. Table 3.1 demonstrates the syntax. In the examples, *A*, *B*, and *C* can be any Rust type name.

Table 3.1: Tylisp S-Expression Syntax (types)

Lisp Syntax	Tylisp Syntax	Rust Type
`()`	`sexpr!{}`	`HNil`
`(A)`	`sexpr!{A}`	`HCons<A, HNil>`
`(A B)`	`sexpr!{A, B}`	`HCons<A,` `  HCons<B, HNil>>`

`(A B C)`	`sexpr!{A, B, C}`	`HCons<A,`
		`  HCons<B,`
		`  HCons<C, HNil>>>`

`(A B . C)`	`sexpr!{A, B; C}`	`HCons<A, HCons<B, C>>`

`((A B) C)`	`sexpr!{{A, B}, C}`	`HCons<`
		`  HCons<A,`
		`  HCons<B, HNil>>,`
		`  HCons<C, HNil>>`

`('A, 'B)`	`sexpr!{@A, @B}`	`HCons<Quote<A>,` `  HCons<Quote<B>, HNil>>`

3.1.2 sexpr_val! Macro

The `sexpr_val!` macro expands to a runtime value composed of a tree of cons cells. Table 3.2 shows several examples, where `a`, `b`, and `c` may be arbitrary Rust expressions. Note that its syntax differs slightly from that of `sexpr!`: There is no shorthand for creating `Quote` values (§3.2.2), and nested lists must be explicitly marked with an `@` sigil. The macro parser needs this sigil to disambiguate between a Rust block used as an expression and a nested list.

Table 3.2: S-Expression Syntax (runtime values)

Lisp Syntax	Tylisp Syntax	Macro Expansion
`()`	`sexpr_val!{}`	`HNil`
`(a)`	`sexpr_val!{a}`	`HCons { head: a, tail: HNil }`
`(a b)`	`sexpr_val!{a, b}`	`HCons { head: a, tail: HCons { head: b, tail: HNil } }`
`(a b c)`	`sexpr_val!{a, b, c}`	`HCons { head: a, tail: HCons { head: b, tail: HCons { head: c, tail: HNil } } }`
`(a b . c)`	`sexpr_val!{a, b; c}`	`HCons { head: a, tail: HCons { head: b, tail; c } }`
`((a b) c)`	`sexpr_val!{@{a, b}, c}`	`HCons { head: HCons { head: a, tail: HCons { head: b, tail: HNil } }, tail: HCons { head: c, tail: HNil } }`

3.1.3 List Trait

`List` indicates that `Self` is a heterogeneous list of finite length.

Prerequisites

`Self` is either the empty list `HNil` or a cons cell whose tail is a `List`.

`List` is implemented for all types which meet these prerequisites. Users may not implement `List` for any custom types.

Associated Constants

`LEN` is a `usize` integer which contaings the number of elements in the list

Associated Types

`Head` is the type of the first element in the list.

`Tail` is a `List` that contains all elements other than the head.

Methods

`split(Self)->(Self::Head, Self::Tail)`

Removes the first item from the list, and returns both pieces

`head(&Self)->&Self::Head`

Returns a reference to the first element of the list

`tail(&Self)->&Self::Tail`

Returns a reference to the tail list

3.1.4 Take Trait

`Take<`*I*`>` indicates that `Self` is an `HList` with an element at location *I*. It is primarily intended for internal Memquery use.

Prerequisites

`Self` must be a `List` of sufficient length. `Take` is automatically implemented for all such lists.

Type Parameters

I is a locator type, either `There<...>` or `Here`. The `Missing` locator type is forbidden, as it represents an item that is not present.

Associated Types

`Taken` is the type of the element at location *I,* and `Remainder` is a list of all remaining elements.

Table 3.3: Sample Take Implementations

`Self`	*`I`*	`Take<`*`I`*`>::Taken`	`Take<`*`I`*`>::Remainder`
`sexpr!{A,B,C}`	`Here`	`A`	`sexpr!{B,C}`
`sexpr!{A,B,C}`	`There<Here>`	`B`	`sexpr!{A,C}`
`sexpr!{A,B,C}`	`There<There<Here>>`	`C`	`sexpr!{B,C}`
`sexpr!{A}`	`There<Here>`	*not implemented*	
`sexpr!{}`	`Here`	*not implemented*	

Methods

`fn take(Self)->(Self::Taken, Self::Remainder)`

Removes the requested element from `Self`.

3.1.5 ListOf Trait

`ListOf<`*X*`>` transforms a heterogeneous list into an iterator of values of type *X*

Prerequisites

`Self` must be a `List`, and every element of `Self` must implement `Into<`*X*`>`. `ListOf<`*X*`>` is automatically implemented for all such lists.

Trait Parameters

X is the type of elements yielded by the `list_of_iter` method.

Associated Types

`ListIter` is the output type of the `list_of_iter` method. Must implement `Iterator<Item=`*X*`>`.

Methods

`list_of_iter(Self)->Self::ListIter`

Consumes `Self` and produces an `Iterator` over its elements

3.1.6 ListOfRefs Trait

`ListOfRefs<`*X*`>` produces an iterator which yields shared references to values of type *X*.

Prerequisites

`Self` must be a `List`, and every element of `Self` must implement `AsRef<X>`. `ListOfRefs<X>` is automatically implemented for all such lists.

Trait Parameters

X is the type of elements that will be yielded from the `iter` method.

Methods

`iter(&Self)->impl Iterator<Item = &X>`

> Returns a Rust iterator over the elements of the list.

`head_ref(&Self)->Option<&X>`

> Returns a reference to the head element of the list. Returns `None` if the list is empty.

`tail_ref(&self)->&dyn ListOfRefs<X>`

> Returns a reference to the tail of the list

3.1.7 HCons Type

`HCons<H,T>` is a cons cell used for constructing heterogeneous lists.

Type Parameters

H is the type of the first element in the list, the head.

T is the type of the remaining list elements, usually `HNil` or another cons cell.

Implemented Traits

Table 3.4: Implemented Traits for HCons<H,T>

Trait	Bounds	Associated Types
`List`	*T*: `List`	`Head = H` `Tail = T`
`ListOf<X>`	*H*: `Into<X>` *T*: `ListOf<X>`	
`ListOfRefs<X>`	*H*: `AsRef<X>` *T*: `ListOfRefs<X>`	
`Copy`	*H*: `Copy` *T*: `Copy`	
`Clone`	*H*: `Clone` *T*: `Clone`	
`Take<I>`	See §3.1.4	

See also §4.4-4.6

3.1.8 HNil Type

`HNil` is a unit type that represents the empty list.

Implemented Traits

Table 3.5: Implemented Traits for HNil

Trait	Associated Types
`List`	`Head = HNil` `Tail = HNil`
`ListOf<X>`	
`ListOfRefs<X>`	
`Copy`	
`Clone`	

See also §4.4-4.6

3.1.9 Locator Types

Tylisp defines 3 locator types to indicate a position within a heterogeneous list:

- `Here` is a unit type that represents the head position of an `HList`.

- `There<I>` is a unit type that represents a position in the tail of an `HList`, where I is a locator type.
- `Missing` is a unit type that indicates there is no valid position.

3.2 Evaluation Model

All Tylisp expressions are purely functional: They are systematically unable to alter their argument types in any way, and can only produce a new output type. Every type *T* that represents a Tylisp expression implements the `Eval` trait, where `Eval::Result` contains the result of evaluating *T*. Any type that has a direct implementation of Eval is considered a Tylisp *atom*:

- Many types are Tylisp *literals*, which evaluate to themselves.
- The `Quote<X>` type evaluates to the type *X*.
- Users are free to implement `Eval` on their own types to describe how they should be interpreted by Tylisp.

Tylisp's computational power lies in its implementation of `Eval` for lists. When evaluating a list, Tylisp first evaluates the head element, which must produce a type *F* that implements the `Call` trait. This specifies the calling convention that will be used to process the remaining elements of the list. At present, there are two conventions defined: `Fun` and `Syn`.

The simplest of these is `Syn`. This indicates that *F* operates directly on the syntax of its arguments, similar to a Lisp macro. The list arguments are not altered, and the function call evaluates to `SynCall::Result`.

The Fun calling convention indicates that *F* is an ordinary function. Each element in the tail of the original list is evaluated, and a list of these results is provided as an argument to F's `FunCall` implementation. The entire function call then evaluates to `FunCall::Result`.

3.2.1 literal! Macro

The `literal!` macro declares that a type is a Tylisp literal. It emits an appropriate implementation of the `Eval` and `LispId` (§3.5.1) traits.

3.2.2 Quote Type

`Quote<X>` is a zero-sized type that evaluates to *X*. It is most commonly produced by using `@` notation inside an `sexpr!` (§3.1.1) call.

Implemented Traits

Table 3.6: Implemented Traits for Quote<X>

Trait	Associated Types
`Eval`	`Result = X`

3.2.3 Eval Trait

`Eval` indicates that `Self` is a well-formed Tylisp expression

Associated Types

`Result` is the output type produced by evaluating `Self`. Note that this is unrelated to the `Result` enum defined by Rust's standard library.

3.2.4 Call Trait

`Call` describes how a `Self` behaves when it appears in a function-call position within a Tylisp expression.

Prerequisites

None. Usually implemented via the `defun!` macro

Associated Types

`Conv` is a marker type that specifies the calling convention to use. Must be either `Fun` or `Syn`.

3.2.5 FunCall Trait

`FunCall<A>` indicates that `Self` is an ordinary Tylisp function which can produce a result from the provided arguments *A*.

Prerequisites

`Self` must implement `Call<Conv=Fun>`. Usually implemented via the `defun!` macro.

Trait Parameters

A is a `List` of the function's arguments, which have already been evaluated once by Tylisp.

Associated Types

`Result` is the output type from executing `Self`. Note that this is unrelated to the `Result` enum defined by Rust's standard library.

3.2.6 SynCall Trait

`SynCall<A>` indicates that `Self` is a syntax function which can produce a result from the provided list *A*.

Prerequisites

`Self` must implement `Call<Conv=Syn>`. Manually implemented.

Trait Parameters

A is the a list of the function's arguments, exactly as they appear in the original Tylisp expression.

Associated Types

`Result` is the output type from the function call. Note that this is unrelated to the `Result` enum defined by Rust's standard library.

3.3 Boolean Logic and Conditionals

Tylisp defines two literals, `True` and `False`, which represent conditions known at compile time. These should not be confused with the Rust literals `true` and `false`, which are both constants of the type `bool`. Tylisp's boolean functions all support short-circuiting: Arguments that are not necessary to determine the result are never evaluated, and do not need to be well-formed.

In order to make interfacing with Rust code easier, Tylisp defines the traits `Pass` and `Fail`, which are implemented for any Tylisp expressions that evaluate to `True` or `False`, respectively.

3.3.1 True and False Types

`True` is a Tylisp literal that represents a Boolean truth value.

`False` is a Tylisp literal that represents a Boolean false value.

3.3.2 If

`If` is a Tylisp syntax function that chooses between two expressions based on a predicate value.

Syntax

`{If,` *Predicate*`,` *Consequent*`,` *Alternate*`}`

Evaluates *Predicate*, and returns the evaluation of either *Consequent* or *Alternate*.

Preconditions

Predicate must evaluate to either `True` or `False`. If *Predicate* evaluates to `True`, *Consequent* must be evaluable. If Predicate evaluates to `False`, *Alternate* must be evaluable. Produces a compile error if these conditions are violated.

Example

```
{If, {EmptyP, @{A,B}},
     @{},
     {Head, @{A,B}}
⟹ A
```

3.3.3 Cond

`Cond` is a Tylisp syntax function that chooses between several expression based on a sequence of predicates.

Syntax

`{Cond,` $\{P_1, C_1\}, \{P_2, C_2\}, \ldots, \{P_n, C_n\}\}$

> Evaluates each of the predicates P_1, P_2, ... P_n in order until one of them P_i evaluates to True. Returns the evaluation of C_i.

Preconditions

P_1 ... P_{i-1} must evaluate to `False`, P_i must evaluate to `True`, and C_i must be evaluable. Produces a compile error if these conditions are violated.

Example

```
{Cond, {False, @A},
       {False, @B},
       {True, @C},
       {False, @D}}
⟹ C
```

3.3.4 And

`And` is a Tylisp syntax function that returns True when all of its arguments evaluate to True.

Syntax

`{And,` $P_1, P_2, \ldots, P_n\}$

> Evaluates each of the predicates P_1, P_2, ... P_n in order until one of them P_i evaluates to `False`. Returns `True` if no False value was found.

Preconditions

P_1 ... P_{i-1} must evaluate to `True` and P_i must evaluate to `False` if it exists. Produces a compile error if these conditions are violated.

Examples

```
{And} ⟹ True
{And, False} ⟹ False
{And, True, True, {EmptyP, @{}}} ⟹ True
{And, True, False, @B} ⟹ False
{And, True, @B, False} ⟹ compile error
```

3.3.5 Or

`Or` is a Tylisp syntax function that returns False when all of its arguments evaluate to False.

Syntax

`{Or,` P_1`,` P_2`, ...,` P_n`}`

> Evaluates each of the predicates P_1, P_2, ... P_n in order until one of them P_i evaluates to `True`. Returns `False` if no True value was found.

Preconditions

P_1 ... P_{i-1} must evaluate to `False` and P_i must evaluate to `True` if it exists. Produces a compile error if these conditions are violated.

Examples

```
{Or} ⟹ False
{Or, False} ⟹ False
{Or, False, False, {EmptyP, @{}}} ⟹ True
{Or, True, False, @A} ⟹ True
{Or, False, @A, True} ⟹ compile error
```

3.3.6 Not

`Not` is an ordinary Tylisp function that inverts the Boolean value of its argument.

Syntax

`{Not,` *Predicate*`}`

Preconditions

Predicate must evaluate to either `True` or `False`. Otherwise, produces a compile error.

Example

```
{Not, {EmptyP, @{}}} ⟹ False
```

3.3.7 Invert

`Invert` is an ordinary Tylisp function that produces an inverted predicate function.

Syntax

`{Invert, `*Predicate*`}`

Preconditions

Predicate must evaluate to a Tylisp function of one argument which returns `True` or `False`

Example

```
{Filter, {Invert, EmptyP}, @{{A}, {}, {B,C}}}
⟹ sexpr!{{A}, {B, C}}
```

3.3.8 Pass Trait

`Pass` indicates that `Self` is a Tylisp expression that evaluates to `True`. `Pass` is automatically implemented for all such expression types.

Example Usage

```
fn at_least_one_empty<A:List, B:List>(a:A, b:B) where
    sexpr!{Or, {EmptyP, @A}, {EmptyP, @B}}: Pass
{ … }
```

3.3.9 Fail Trait

`Fail` indicates that `Self` is a Tylisp expression that evaluates to `False`. `Fail` is automatically implemented for all such expression types.

Example Usage

```
fn not_both_empty<A:List, B:List>(a:A, b:B) where
    sexpr!{And, {EmptyP, @A}, {EmptyP, @B}}: Fail
{ … }
```

3.4 List Manipulation

Tylisp defines a number of functions for manipulating heterogeneous lists.

3.4.1 EmptyP

`EmptyP` is an ordinary Tylisp function that returns `True` if its argument is the empty list.

Syntax

`{EmptyP, L}`

Preconditions

L must evaluate to a type which implements `List`. Produces a compile error otherwise.

Examples

```
{EmptyP, @{}} ⟹ True
{EmptyP, @{A}} ⟹ False
```

3.4.2 Head

`Head` is an ordinary Tylisp function that returns the `head` field a cons cell.

Syntax

`{Head, L}`

Preconditions

L must evaluate to a cons cell. Produces a compile error otherwise.

Examples

```
{Head, @{}} ⟹ compile error
{Head, @{A}} ⟹ A
{Head, @{A, B}} ⟹ A
{Head, @{A; B}} ⟹ A
```

3.4.3 Tail

`Tail` is an ordinary Tylisp function that returns the `tail` field of a cons cell.

Syntax

`{Tail, L}`

Preconditions

L must evaluate to a cons cell. Produces a compile error otherwise.

Examples

```
{Tail, @{}} ⟹ compile error
{Tail, @{A}} ⟹ sexpr!{}
{Tail, @{A, B}} ⟹ sexpr!{B}
{Tail, @{A; B}} ⟹ B
```

3.4.4 Cons

`Cons` is an ordinary Tylisp function that constructs a cons cell from its arguments.

Syntax

`{Cons, `*Head*`, `*Tail*`}`

Preconditions

Head and *Tail* must be evaluable.

Examples

```
{Cons, @A, @B} ⟹ sexpr!{A; B}
{Cons, @A, @{B,C}} ⟹ sexpr!{A, B, C}
{Cons, @{A, B}, @C} ⟹ sexpr!{{A, B}; C}
{Tail, @{A, B}, @{C, D}} ⟹ sexpr!{{A, B}, C, D}
```

3.4.5 Map

`Map` is an ordinary Tylisp function which applies a Tylisp function to every element of a list.

Syntax

`{Map, `*Func*`, `*List*`}`

Preconditions

List must evaluate to a type which implements `List`.

For every element *x* in *List*, the expression `{Func, @`*x*`}` must be evaluable. Produces a compile error if this is violated.

Example

```
{Map, EmptyP, @{{A,B}, {}, {C}}}
⟹ sexpr!{False, True, False}
```

3.4.6 Filter

`Filter` is an ordinary Tylisp function which selects elements from a list based on a predicate function

Syntax

`{Filter, `*Func*`, `*List*`}`

Preconditions

List must evaluate to a type which implements `List`.

For every element *x* in *List*, the expression `{Func, @x}` must evaluate to either `True` or `False`. Produces a compile error if this is violated.

Example

```
{Filter, {Partial, Contains, @{A, B}},
         @{C, B, A}}
⟹ sexpr!{C, A}
```

3.4.7 Collate

`Collate` is an ordinary Tylisp function which divides a list's elements into two lists based on a predicate function.

Syntax

`{Collate, `*Func*`, `*List*`}`

Preconditions

List must evaluate to a type which implements `List`.

For every element *x* in *List*, the expression `{Func, @x}` must evaluate to either `True` or `False`. Produces a compile error if this is violated.

Example

```
{Filter, {Partial, Contains, @{A, B}},
         @{C, B, A}}
⟹ sexpr!{{C, A}, {B}}
```

3.4.8 CollatedBy Trait

`CollatedBy<`*Expr*`>` indicates that `Self` is a heterogeneous list that can be filtered by the Tylisp expression *Expr*.

Trait Parameters

Expr is a Tylisp expression that evaluates to a Tylisp function

Prerequisites

The Tylisp expression `{Collate, `*Expr*`, @Self}` must be evaluable. `CollatedBy<`*Expr*`>` is automatically implemented for all lists that satisfy this prerequisite.

Associated Types

`Passed` is a `List` of the elements for which *Expr* evaluates to True

`Failed` is a `List` of the elements for which *Expr* evaluates to False

Methods

`collate(Self)->(Self::Passed, Self::Failed)`

Divides `Self` into two lists according to the predicate *Expr*.

```
CollatedBy::<sexpr!{Partial, Is, @A}>
    ::collate(sexpr_val!{A(1), B(2), A(3), C(4)})
⟹
(sexpr_val!{A(1), A(3)}, sexpr_val!{B(2), C(4)})
```

3.4.9 BuildList

`BuildList` is an ordinary Tylisp function which returns a list of its arguments. This differs from simply writing a quoted list in that the arguments are evaluated before the list is built.

Syntax

`{BuildList, @A, @B, ... } ⟹ sexpr!{A, B, ... }`

Preconditions

Every argument must be evaluable.

3.4.10 Reverse

`Reverse` is an ordinary Tylisp function which reverses the order of elements in a list.

Syntax

`{Reverse,` *L*`}`

Preconditions

L must evaluate to a type which implements `List`

Example

```
{Reverse, @{A, B, C}} ⟹ sexpr!{C, B, A}
```

3.4.11 Any

`Any` is an ordinary Tylisp function which returns `True` if any elements in a list satisfy the given predicate function, or `False` if all elements fail the predicate function. It is unrelated to the `Any` trait defined in Rust's standard library.

Syntax

`{Any,` *Predicate*`,` *List*`}`

Preconditions

List must evaluate to a type that implements `List`.

Predicate must evaluate to a Tylisp function which takes a single argument.

Given *List* elements x_1, x_2, ... x_i ... x_n, where {*Predicate*, @x_i} evaluates to `True`, {*Predicate*, @x_k} must evaluate to `False` for all $k < i$. The remaining elements x_{i+1} ... x_n are irrelevant and do not need to be valid arguments to *Predicate*.

If there is no i for which {*Predicate*, @x_i} evaluates to `True`, {*Predicate*, @x_k} must evaluate to `False` for all $k \leq n$.

Example

```
{Any, {Partial, Is, @A}, @{C, B, A, D}} ⟹ True
{Any, {Partial, Is, @A}, @{C, B, D}} ⟹ False
```

3.4.12 All

`All` is an ordinary Tylisp function which returns `True` if all elements in a list satisfy the given predicate function, or `False` if any element fails the predicate function.

Syntax

{`All`, *Predicate*, *List*}

Preconditions

List must evaluate to a type that implements `List`.

Predicate must evaluate to a Tylisp function which takes a single argument.

Given *List* elements x_1, x_2, ... x_i ... x_n, where {*Predicate*, @x_i} evaluates to `False`, {*Predicate*, @x_k} must evaluate to `True` for all $k<i$. The remaining elements x_{i+1} ... x_n are irrelevant and do not need to be valid arguments to *Predicate*.

If there is no i for which {*Predicate*, @x_i} evaluates to `False`, {*Predicate*, @x_k} must evaluate to `True` for all $k \leq n$.

Example

```
{All, EmptyP, @{{}, {}, {A, B}}} ⟹ False
```

3.4.13 FindPred

`FindPred` is an ordinary Tylisp function that returns a locator type (§3.1.9) which indicates the position of the first element in a list that satisfies the given predicate.

Syntax

{`FindPred`, *Predicate*, *List*}

Preconditions

List must evaluate to a type which implements `List`.

Given *List* elements x_1, x_2, ... x_i ... x_n, where {*Predicate*, x_i} evaluates to `True`, {*Predicate*, x_k} must evaluate to `False` for all $k<i$.

Examples

```
{FindPred, EmptyP, @{{A}, {}, {}}} ⟹ There<Here>
{FindPred, {Invert, EmptyP}, @{{}, {}, {}}} ⟹ Missing
```

3.4.14 Concat

`Concat` is an ordinary Tylisp function that concatenates two lists.

Syntax

`{Concat, A, B}`

Preconditions

A and *B* must both evaluate to types which implement `List`

Example

```
{Concat, @{A, B}, @{C, D}} ⟹ sexpr!{A,B,C,D}
```

3.5 Type Comparison and Set Algebra

Part of Rust's forward compatibility strategy is to allow developers to add a new trait implementation to an existing type without breaking existing code. To ensure this remains possible in most cases, there is no mechanism to bound implementations on the absence of a trait. This also extends to comparing two types for equality: It is possible to enforce that two generic parameters refer to the same type, but impossible to enforce that they refer to different types. This introduces a difficulty for implementing set algebra within the type system: When performing a set insert operation, for example, it is necessary to verify that the type of the new element is not already present in the set.

One way to get around this limitation is to operate only on a universe *U* of types that is completely known in advance. Using this scheme, testing that a type is not in a set *S* is equivalent to testing that it is in the set $U \setminus S$. The `typenum` crate uses this principle to implement types that represent the domain of nonzero integers[6] [13]. Each integer is represented as a heterogeneous list of `True` and `False` types, which directly correspond to its binary representation.

In order to support set operations on arbitrary types, Tylisp requires that all types in a set implement `LispId`, which holds a unique ID encoded as one of `typenum`'s numeric types. To facilitate the creation of these IDs, the `uuid_new_v4!` macro will generate a random UUID and encode it as 128-bit `typenum` integer.

6 The typenum crate also implements representations of other numeric domains, but they are not relevant to this project.

3.5.1 LispId Trait

The `LispId` trait associates a unique integer with `Self`, which allows Tylisp to perform operations that depend on whether or not `Self` is equal to some other type.

Prerequisites

`Id` must be a `typenum` integer type. No two `LispId` implementations may specify the same `Id`. This is not mechanically enforced. All implementations provided by Tylisp or Memquery are version 4 (random data) UUIDs [14].

Associated Types

`Id` is a type-level integer that uniquely identifies `Self`.

3.5.2 uuid_new_v4! Macro

The `uuid_new_v4!` macro evaluates to a `typenum` integer type which represents a newly-generated version 4 UUID.

Syntax

```
type Id = uuid_new_v4!{};
```

3.5.3 Is

`Is` is an ordinary Tylisp function which compares its arguments for type equality. This function cannot be used to compare lists, as they do not implement `LispId`; use `DifferP` for that instead.

Syntax

`{Is, A, B}`

Returns `True` if *A* and *B* are the same type

Preconditions

A and *B* must evaluate to types which implement the trait `LispId`.

3.5.4 DifferP

`DifferP` is an ordinary Tylisp function which compares two lists for equality.

Syntax

`{DifferP, A, B}`

Returns `True` if *A* and *B* are the same length and every element in *A* is the same type as the corresponding element in *B*.

Preconditions

A and *B* must both evaluate to types which implement `List` and every element in both *A* and *B* must implement `LispId`.

Examples

```
{DifferP, @{}, @{}} ⟹ False
{DifferP, @{A}, @{B}} ⟹ True
{DifferP, @{A}, @{A, B}} ⟹ True
```

3.5.5 Contains

`Contains` is an ordinary Tylisp function that checks for the presence of a specific element inside a list.

Syntax

`{Contains,` *List*`,` *Item*`}`

Returns `True` if *Item* is an element of *List*

Preconditions

List must evaluate to a type which implements `List`.

Item must evaluate to a type which implements `LispId`.

Every element of *List* must implement `LispId`.

Examples

```
{Contains, @{A, B, C}, @A} ⟹ True
{Contains, @{A, B, C}, @D} ⟹ False
```

3.5.6 SupersetP

`SupersetP` is an ordinary Tylisp function that checks whether the elements of one list are a superset of the elements of another list.

Syntax

`{SupersetP,` *A*`,` *B*`}`

Returns `True` if all elements of *B* are present in *A*.

Preconditions

A and *B* must both evaluate to types which implement `List`. All elements of *A* and *B* must implement `LispId`.

Examples

```
{SupersetP, @{A, B, C}, @{C, A}} ⟹ True
{SupersetP, @{A, B, C}, @{}} ⟹ True
{SupersetP, @{A, B, A}, @{B, B}} ⟹ True
{SupersetP, @{A, B], @{B, A}} ⟹ True
{SupersetP, @{A, B], @{A, C}} ⟹ False
```

3.5.7 SubsetP

`SubsetP` is an ordinary Tylisp function that checks whether the elements of one list are a subset of the elements in another.

Syntax

`{SubsetP, `*A*`, `*B*`}`

> Returns `True` if all elements of *A* are present in *B*. Equivalent to `{SupersetP, `*B*`, `*A*`}`

Preconditions

Both *A* and *B* must evaluate to types which implement `List`. All elements of both *A* and *B* must implement `LispId`.

3.5.8 Without

`Without` is an ordinary Tylisp function that removes all instances of a type from a list.

Syntax

`{Without, `*Item*`, `*List*`}`

> Returns a list that contains all elements of *List* which are not of type *Item*.

Preconditions

List must evaluate to a type which implements `List`.

Item must evaluate to a type which implements `LispId`.

Every element of *List* must implement `LispId`.

Example

```
{Without, @A, @{A, B, A, C}} ⟹ sexpr!{B, C}
```

3.5.9 Find

`Find` is an ordinary Tylisp function that locates the first instance of a type in a list

Syntax

`{Find, `*Needle*`, `*Haystack*`}`

> Returns a locator type (§3.1.9) which indicates the position of *Needle* within *Haystack*.

Preconditions

Haystack must evaluate to a type which implements `List` and every element of *Haystack* must implement `LispId`.

Needle must evaluate to a type which implements `LispId`.

Examples

```
{Find, @A, @{A, B, A C}} ⟹ Here
{Find, @A, @{B, A, C}} ⟹ There<Here>
{Find, @A, @{B, C}} ⟹ Missing
```

3.5.10 Union

`Union` is an ordinary Tylisp function that computes the union of two sets.

Syntax

`{Union, `*A*`, `*B*`}`

> Prepends the elements of *A* which are not present in *B* to *B*. Note that if either *A* or *B* contains internally duplicated elements, those elements may be duplicated in the result.

Preconditions

Both *A* and *B* must evaluate to types which implement `List`. Every element of both *A* and *B* must implement `TypeId`.

Examples

```
{Union, @{A, B}, @{B, C}} ⟹ sexpr!{A, B, C}
{Union, @{A, C}, @{B, D}} ⟹ sexpr!{A, C, B, D}
{Union, @{C, A}, @{A, B, C}} ⟹ sexpr!{A, B, C}
{Union, @{A}, @{A, B, A}} ⟹ sexpr!{A, B, A}
{Union, @{B, B}, @{A, C}} ⟹ sexpr!{B, B, A, C}
{Union, @{A, A}, @{A, C}} ⟹ sexpr!{A, C}
```

3.5.11 Intersect

`Intersect` is an ordinary Tylisp function which computes the intersection of two sets.

Syntax

`{Intersect, A, B}`

Returns a list of the elements of B that are also elements of A. Note that if B contains any duplicate elements, they will be duplicated in the result.

Preconditions

Both *A* and *B* must evaluate to types which implement `List`. Every element of both *A* and *B* must implement `LispId`.

Examples

```
{Intersect, @{A, B}, @{B, C}} ⟹ sexpr!{B}
{Intersect, @{A, B}, @{B, C, A}} ⟹ sexpr!{B, A}
{Intersect, @{A, B}, @{A, C, A}} ⟹ sexpr!{A, A}
{Intersect, @{A, B}, @{C, D}} ⟹ sexpr!{}
```

3.5.12 Remove

`Remove` is an ordinary Tylisp function which calculates the difference between two sets.

Syntax

`{Remove, A, B}`

Returns a list containing all elements of *B* that are not elements of *A*. Note that if *B* contains duplicate elements, then they will also be duplicated in the result

Preconditions

Both *A* and *B* must evaluate to types which implement `List`. Every element of both *A* and *B* must implement `LispId`.

Examples

```
{Remove, @{A, B}, @{B, C}} ⟹ sexpr!{C}
{Remove, @{A, B}, @{B, C, A}} ⟹ sexpr!{C}
{Remove, @{B, C}, @{A, C, A}} ⟹ sexpr!{A, A}
{Remove, @{A, B}, @{C, D}} ⟹ sexpr!{C, D}
```

3.5.13 SetInsert

`SetInsert` is an ordinary Tylisp function which inserts an element into a set.

Syntax

`{SetInsert, Item, List}`

Prepends *Item* to *List* iff *List* contains no elements of type *Item*.

Preconditions

List must evaluate to a type which implements `List`, every element of *List* must implement `LispId`.

Item must evaluate to a type which implements `LispId`.

Examples

```
{SetInsert, @A, @{C, A, B}} ⟹ sexpr!{C, A, B}
{SetInsert, @A, @{B, C}} ⟹ sexpr!{A, B, C}
```

3.6 Defining Functions

One notable absence from Tylisp is any kind of variable-binding form, such as let or lambda. There are a few function combinators, such as `Invert` (§3.3.7) and `Partial`, but most Tylisp functions are defined via the `defun!` Rust macro, outside of any Tylisp expression.

3.6.1 defun! macro

Defines a Rust type that acts as an ordinary Tylisp function.

Syntax

```
defun!{ Name {
    (generics) {args} => expr;
    (generics) {args} => expr;
    ...
}}
```

Name is the function to be defined. Each arm defines a `FunCall` (§3.2.5) implementation for the given generic types and argument list; Rust will use pattern matching to choose the appropriate arm at each callsite. Each *expr* is a Tylisp function call which descibes the function result.

Example: Recursion

A function to turn a list into a list of 2-tuples can be defined like this:

```
defun!{ Pairs {
    (A, B, Tail) {{A, B; Tail}} => {Cons, @(A, B),
                                    {Pairs, @Tail}};
    ()           {{}          } => {Ret, @{}};
}}
```

Each of the two arms define a function that accepts a single argument. The first arm defines the behavior for lists of at least two elements, *A* and *B*. It packages them into a 2-tuple, and makes a recursive call to process the rest of the list. The second arm describes the base case, where the argument is the empty list.

If a user attempts to pass a list with an odd length to this function, such as `{X, Y, Z}`, it will produce a compile error due to a missing implementation for length-1 lists.

```
{Pairs, @{X, Y, Z, W}} ⟹ sexpr!{(X, Y), (Z, W)}
{Pairs, @{X, Y, Z}} ⟹ Compile error
```

The argument list can be composed of arbitrary types, not just ones named as generic parameters:

```
defun!{ Key {
    (K,V) { BTreeMap<K,V> } => {Ret, @K};
    (K,V) { HashMap<K,V>  } => {Ret, @K};
    (T)   { Vec<T>        } => {Ret, @usize};
}}
```

Example: Generic Bounds

It is also possible to specify Rust type bounds for the generic parameters. For example, this function accepts one argument, which must be a reference type:

```
defun!{ Target {
    (Ptr: Deref) {Ptr} => {Ret, @Ptr::Target};
}}

{Target, @Rc<usize>} ⟹ usize
```

3.6.2 Ret

`Ret` is an ordinary Tylisp function that returns its argument.

Syntax

`{Ret, @A} ⟹ A`

Preconditions

The argument must be evaluable.

3.6.3 Partial

`Partial` is a Tylisp syntax function that implements partial application of a function

Syntax

`{Partial,` $F, A_1, A_2, \ldots, A_n$`}`

> Constructs a new Tylisp function F_2 such that the expression `{`F_2`,` $B_1, B_2, \ldots B_m$`}` is equivalent to `{`F`,` $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$`}`

Example

```
{Map, {Partial, SetInsert, @A}, @{{}, {A, B}, {B, C}}}
⟹ sexpr!{{A}, {A, B}, {A, B, C}}
```

4 Memquery: Relational Algebra in Rust

4.1 Overview

Memquery is a framework for managing a program's internal data, based on the principles of relational algebra. It is designed to reduce the coupling between three distinct programming roles:

1. *Application programmers* who are primarily concerned with the correctness of a single feature or use case of a program.

2. *Architects* who are primarily concerned with the overall performance and maintainability of a program.

3. *Library authors* who are primarily concerned with inventing new data organization schemes that can be used in multiple programs.

4.1.1 Data Model

The smallest unit of data in Memquery is the *column value*, which stores a single atomic value. These are represented by single-field structures which implement the `Col` trait (§4.3.3). The particular set of column types available are unique to each program, according to the needs of its data model.

Individual facts are modeled as a set of column values that are related to each other. These facts are represented in Memquery by types that implement the Record trait (§4.5.1). The set of column types contained in a record is known as its *header*. All column types also implement `Record`, as do tuples of records[7]. It is also possible for programs to define custom `Record` types.

A collection of records with uniform type is known as a relation, represented by the `Relation` trait (§4.2.1). This trait provides facilities to query the contained records, and the related traits Insert (§4.9.4) and Delete (§4.9.5) provide a common interface for modifying the contents of relations.

Query results are always Record types that borrow their column data from the relation being queried, so that the column references may outlive the returned record. The `ExternalRecord<'a>` trait (§4.5.2) represents this property; it provides methods to access longer-lived references than are available from the `Record` trait.

7 A tuple only implements Record if its component records have disjoint headers.

4.1.2 For Application Programmers

Application programmers need to store data into and retrieve data from the relations that have been defined by the architect. Because all Memquery relations present a uniform interface for these tasks, the programmer does not need to pay much attention to the particular relation type.

Querying a Relation

To illustrate a typical query, consider a relation vendors that contains, among others, the columns `VendorId` and `VendorName`. To print a list of the vendors in alphabetical order:

```
for (VendorId(id), VendorName(name))
 in vendors.by_ref()
           .order_by::<sexpr!{Asc<VendorName>}>()
           .iter_as()
{
    println!("{:4} {}", id, name);
}
```

This makes use of three different methods from the Relation trait:

1. `by_ref()` prevents the vendors relation from being consumed by the query
2. `order_by::<Asc<VendorName>>()` requests the records to be presented in ascending order by name
3. `iter_as()` will iterate over the query results. It has a polymorphic return type, which Rust infers based on the pattern `(VendorId(id), VendorName(name))`

Joining Relations

Suppose there is another relation, suppliers, which contains `(PartId, VendorId)` pairs to represent the vendors capable of supplying various parts. We can extend the example above to only list vendors that supply a particular part (#42) like this:

```
for (VendorId(id), VendorName(name))
 in vendors.by_ref()
           .join(suppliers.by_ref())
           .where_eq(PartId(42))
           .order_by::<sexpr!{Asc<VendorName>}>()
           .iter_as()
{
    println!("{:4} {}", id, name);
}
```

Here, there are two additional method calls in the chain of query adapters:

1. `join(suppliers.by_ref())` constructs the natural join of the two relations
2. `where_eq(PartId(42))` selects only those records that pertain to part #42

Inserting Records

The `Insert` (§4.9.4) trait defines two methods for inserting records into a relation: `insert()` adds a single record and `insert_multi()` adds several. These operations are atomic: If any of the insertions fail, no changes are made to the relation.

```
vendors.insert(
    (VendorId(42),
     VendorName(String::from("Acme, Inc."))
    )
).unwrap();

suppliers.insert_multi(vec![
    (VendorId(42), PartId(3)),
    (VendorId(42), PartId(5)),
    (VendorId(37), PartId(5))
]).unwrap();
```

Removing Records

If a relation type implements the `Delete` trait (§4.9.5), it supports removing records. The `Relation::truncate` method will delete all records from these relations. Calling `truncate` on a query adapter will only delete the records that are visible through the adapter, and leave all other records unaffected. For example, to delete all supplier records for vendor #42:

```
vendors.by_mut()
       .where_eq(VendorId(42))
       .truncate();
```

Updating Records

There is no way to modify records in-place. Instead, they need to be removed, modified, and then re-inserted. To make this robust against additional columns being added to the relation, the associated type `Relation::Cols` can be used as a temporary record type that contains all of the relation's columns.

```
#[derive(Copy,Clone,Ord,PartialOrd,Eq,PartialEq)]
pub enum OrderStatus {
    Approved,
    Sent,
    Received,
}

col!{ pub PartId: u32 }
col!{ pub Quantity: u32 }
col!{ pub Status: OrderStatus }

type Orders = Vec<(PartId, Quantity, OrderStatus)>;

fn send_approved(orders: &mut Orders) {
    use OrderStatus::*;

    // Find orders that have been approved
    let mut approved = orders.by_mut()
                             .where_eq(Status(Approved));

    // Store a copy of the approved orders
    let mut updates: Vec<Orders::Cols> =
        approved.iter_as().collect();

    // Make necessary modifications
    for order in &mut updates {
        **(order.col_mut::<Status>()) = Sent;
    }

    // Remove the old records
    approved.truncate();

    // Insert the updated records
    orders.insert_multi(updates).unwrap();
}
```

Transactions

Sometimes, it is necessary to perform multiple fallible operations atomically, such that no changes occur unless they all succeed. Inserts, for example, can fail due to constraint violations. Memquery provides a transaction system to handle this situation (§4.9); the `Insert` and `Delete` traits provide the associated functions `insert_op` and `delete_op` to generate revertable operations which can be applied in a transaction.

Suppose the programmer wishes to add a new vendor and the parts that vendor can supply to our example database, but only if all of the records can be inserted successfully. One way to write this is:

```
type Vendors = ...;
type Suppliers = ...;

// Add new vendor
let vendor_txn = Transaction::start(vendors)
    .apply(Vendors::insert_op(
        (VendorId(vendor_id), VendorName(name))
    );

// Add supplier records
let supplier_txn = Transaction::start(suppliers)
    .apply_multi( vendor_part_ids.map(
        |part_id| Suppliers::insert_op(
            (VendorId(vendor_id), PartId(part_id))
        )
    ));

match (vendor_txn.inspect(), supplier_txn.inspect()) {
    (Some(_), Some(_)) => {
        // All inserts were successful
        vendor_txn.commit().unwrap();
        supplier_txn.commit().unwrap();
    }
    _ => {
        // Something went wrong
        vendor_txn.revert();
        supplier_txn.revert();
    }
}
```

4.1.3 For Architects

The architect's primary job in a Memquery-based program is to define the schema that the application programmers will use. This consists of declaring the columns and relations that the program will use.

Declaring Columns

Columns are declared via the `col!` macro (§4.3.2). Each column definition consists of an optional visibility specifier, the column name, and a Rust type that describes the domain of values that can be stored within the column. Each invocation of the `col!` macro defines a new type which represents a single value from that column. For example, this defines two columns, `PartId` which stores an unsigned 32-bit integer and `PartName` which stores a text string:

```
col!{ pub PartId: u32 }
col!{ pub PartName: String }
```

Because each column is represented by a type, column names can be reused as long as each of the duplicate names is defined in a separate module. This can be useful to organize a columns into related groups. In the example below, `part::Id` and `project::Id` are completely distinct columns; application code that deals only with parts can import the definitions from the part module to use the shorter name.

```
pub mod part {
    col!{ pub Id: u32 }
    col!{ pub Name: String }
}

pub mod project {
    col!{ pub Id: u32 }
    col!{ pub Name: String }
}
```

Declaring Relations

The most basic relation types provided by Memquery are `Vec<T>` and `Option<T>` from Rust's standard library. `Vec<T>` stores zero or more records of type *T*, and `Option<T>` stores zero or one record of type *T*. The header for each relation is implicitly defined by the header of the records which it contains. In many cases, the record type stored in a relation can be simply a tuple of column types. The most convenient way to define a schema is to define a singleton structure which contains all of the schema's relations as fields. A database that contains one relation with the columns `PartId` and `PartName` can be defined like this:

```
col!{ pub PartId: u32 }
col!{ pub PartName: String }

#[derive(Default)]
pub struct DatabaseA {
    parts: Vec<(PartId, PartName)>
}
```

This definition has some drawbacks: Every query will need to perform a sequential scan of the parts relation, and there can be multiple records with the same PartId. Correcting these deficiencies requires adding an index to the parts relation.

In Memquery, indices are relations that are composed of subrelations. A `BTreeIndex<K,R>` (§4.8.1), for example, contains several instances of the relation type *R*, one for each unique value of the column *K*. This definition, for example, supports all of the same queries as the definition above:

```
#[derive(Default)]
pub struct DatabaseB {
    parts: BTreeIndex<PartId,
                      Option<(PartId, PartName)>>
}
```

Here, the original relation `Vec<...>` has been replaced with `Option<...>`. This only allows a single `PartName` to be stored for each unique `PartId`; any attempt to insert a duplicate id will be rejected. Also, instead of a sequential scan, the `BTreeIndex` will perform a direct lookup for any query that specifies an explicit `PartId` value. These benefits come with some extra costs for other operations, however: Instead of a single continuous memory allocation, the records will be stored in various BTree nodes scattered throughout the heap, potentially harming performance when iterating through all of the records.

Queries that specify a part's name instead of its id will still require a sequential scan of all records. `RedundantIndex<`K_2`,`K_1`,`R`>` (§4.8.2) can be used to provide a secondary index on the PartName column:

```
#[derive(Default)
pub struct DatabaseC {
    parts: RedundantIndex<PartName, PartId,
               BTreeIndex<PartId,
                   Option<(PartId, PartName)>>>
}
```

Unlike `BTreeIndex`, `RedundantIndex` contains only a single instance of its declared subrelation R, in this case `BTreeIndex<...>`. Alongside this, it stores a mapping from K_2 (`PartName`) to K_1 (`PartId`) values. When a query specifies an explicit part name, the corresponding ids are retrieved from this mapping and used to retrieve the correct records from the `BTreeIndex`. Inserting new records into this structure, however, is roughly twice as expensive as inserting them into `DatabaseB`: Each record must be inserted into two separate index structures instead of just one.

As the parts relation in all three of these examples contains the same set of columns, most application code that works with one version will also work correctly with any of the others. If a new column is added to a relation, most query code will continue to work correctly, but any insertion code will need to provide a value for the new column.

4.1.4 For Library Authors

Library authors may wish to define new relation types that integrate into the rest of the Memquery ecosystem. This is probably the most complicated Memquery programming task. The complexity of this task is, in part, how Memquery is able to present a simplified, but still performant, interface to architects and application programmers.

All of Memquery's machinery for executing queries is contained in a blanket implmentation of the `Queryable<'a, `Q`>` trait (§4.2.5). Library authors need to be familiar with how `Queryable` works in order to make their extension types interact with the rest of the Memquery ecosystem. Also, unlike application programmers, library authors will usually call `Queryable::query()` directly instead of constructing adapters when they need to retrieve records from a relation.

Every query request is represented by a type that implements the `QueryRequest` trait (§4.6.1), which specifies a set of selection filters and a presentation order for records. Given a query request q of type Q, `Queryable<'a, Q>::query(&'a self, q)` performs a three-step process:

1. Obtain a query plan P by executing the Tylisp expression
   `{<Self as RelationImpl>::Planner, @&'a Self, @Q}`

2. Construct an instance p of type P by calling
   `<P as QueryPlanImpl>::prepare(&'a self, q)`

3. Obtain an iterator of result records by calling
   `<P as IntoIterator>::into_iter(p)`

`Queryable<'a, Q>` is automatically implemented for all types where this process passes the type checker and produces an iterator that yields items of type `<Self as QueryOutput<'a>>::QueryRow`. The primary task of a library author, then, is to provide implementations of these traits, so that their extension type implements `Queryable` for every possible `QueryRequest` type Q.

Defining a Query Adapter

To illustrate the process, consider a relation adapter that discards records that contain a duplicated value in a named column. The first step is to define a structure to represent the new adapter:

```
struct Unique<C,R>{
    rel: R,
    col: PhantomData<C>
}

impl<C:Col+Ord, R:RelationImpl> Unique<C,R> {
    fn new(rel:R)->Self {
        Unique { rel, col: PhantomData }
    }
}
```

The `RelationImpl` and `QueryOutput` traits (§4.2.3,4) describe the properties of the relation. In this case, it has the same header as its argument relation R, and will be returning the same record type. Because it delegates all queries to R, it also provides the same set of indexed (fast) columns. As it will be inspecting the value of the column C, it must be present in R.

```
impl<C:Col,R> RelationImpl for Unique<C,R> where
    R: RelationImpl,
    R::Cols: HasCol<C>
{
    type Cols = R::Cols;
    type FastCols = R::FastCols;
    type Planner = UniquePlanner;
}

impl<'a,C:Col,R> QueryOutput<'a> for Unique<C,R> where
    R:QueryOutput<'a, Cols=Self::Cols>,
    Self:RelationImpl
{
    type QueryRow = R::QueryRow;
}
```

We also need to define a query plan that can iterate over the results. This will require a set to keep track of the C values that we've already seen and the iterator over R's query results:

```
struct UniquePlan<'a,C:Col,Out>
{
    seen: BTreeSet<&'a C>,
    iter: Box<dyn Iterator<Item=Out> + 'a>
}

impl<'a,C:Col+Ord,Out> Iterator for UniquePlan<'a,C,Out> where
    Out: ExternalRecord<'a>,
    Out::Cols: HasCol<C>
{
    type Item = Out;
    fn next(&mut self)->Option<Self::Item> {
        for result in &mut self.iter {
            if self.seen.insert(result.ext_col_ref()) {
                return Some(result)
            }
        }
        None
    }
}
```

To associate this plan with `Unique`, we need to define the `UniquePlanner` Tylisp function. Because all queries will use the same plan, it can simply return the appropriate `UniquePlan` type:

```
defun!{UniquePlanner {
    ('a, C:Col, R:QueryOutput<'a>, Q) { &'a Unique<C,R>, Q }
    => { Ret, @UniquePlan<'a, C, R::QueryRow> };
}}
```

The `QueryPlanImpl` trait (§4.2.6) then describes how to construct a `UniquePlan` instance for a given relation and query:

```
impl<'a,C:Col+Ord,R,Q> QueryPlanImpl<'a,Unique<C,R>,Q>
for UniquePlan<'a,C,R::QueryRow> where
    Q: QueryRequest + 'a,
    R: Queryable<'a, Q>,
{
    fn prepare(r: &'a Unique<C,R>, q:Q)->Self {
        UniquePlan {
            seen: BTreeSet::new(),
            iter: Box::new(r.rel.query(q))
        }
    }
}
```

Finally, we need to define how to iterate over all of `Unique`'s records by implementing `IntoIterator` for `&Unique`:

```
impl<'a,R,C> IntoIterator for &'a Unique<C,R> where
    R: Relation<'a>,
    C: Col + Ord,
    R::Cols: HasCol<C>,
    &'a R: IntoIterator<Item = R::QueryRow>
{
    type IntoIter = UniquePlan<'a,C,R::QueryRow>;
    type Item = R::QueryRow;
    fn into_iter(self)->Self::IntoIter {
        UniquePlan {
            seen: BTreeSet::new(),
            iter: Box::new(
                      <&R as IntoIterator>::into_iter(&self.rel)
                  )
        }
    }
}
```

This completes the implementation of `Unique`. It can contain any Memquery relation, and will use that relation's query planner to generate candidate results. Unique can also be used as a source relation for any other query adapter:

```
col!{A:usize}
col!{B:usize}

let rel = vec![(A(1), B(1)), (A(2), B(2)), (A(1), B(3))];
Unique::<A,_>::new(rel)
              .iter_as()
              .collect::<Vec<B>>();

⟹ vec![B(1), B(2)]
```

Defining a Storage Relation

In addition to custom operations, a libraray author may want to implement a new way to store records. This must support inserting and deleting records in addition to queries. To illustrate this, consider a relation that contains both small, frequently

accessed columns and large, infrequently accessed columns. It may be possible to improve the cache efficiency of queries that access the small columns by storing them in a separate allocation from the large columns. As before, start by defining a structure which will hold all of the relation's data:

```
struct SplitVec<L,R> {
    left: Vec<L>,
    right: Vec<R>
}
```

As this structure has no way to speed up particular queries, there is no need to define a custom query planner. Instead, the sequential-scan `FallbackPlanner` (§4.2.8) can be used instead:

```
impl<L,R> RelationImpl for SplitVec<L,R> where
    L: Record,
    R: Record,
    (L,R): Record
{
    type Cols = <(L,R) as Record>::Cols;
    type FastCols = sexpr!{};
    type Planner = FallbackPlanner;
}
```

For the `FallbackPlanner` to work, `QueryOutput` and `IntoIterator` must be defined to iterate over all the records in the relation. In this case, the output records are pairs of references:

```
impl<'a,L,R> QueryOutput<'a> for SplitVec<L,R> where
    Self: RelationImpl + 'a,
    (&'a L, &'a R): ExternalRecord<'a, Cols=Self::Cols>
{
    type QueryRow = (&'a L, &'a R);
}

impl<'a,L,R> IntoIterator for &'a SplitVec<L,R> {
    type Item = (&'a L, &'a R);
    type IntoIter = Zip<std::slice::Iter<'a, L>,
                        std::slice::Iter<'a, R>>;
    fn into_iter(self)->Self::IntoIter {
        self.left.iter().zip(self.right.iter())
    }
}
```

At this point, `SplitVec` instances support all query requests, but there is no way to either construct an instance or insert new rows. Storage relations should implement the `Default` trait to construct empty instances:

```
impl<L,R> Default for SplitVec<L,R> {
    fn default()->Self {
        SplitVec {
            left: vec![],
            right: vec![]
        }
    }
}
```

Next, define a `RevertableOp` (§4.9.1) for inserting an individual record and a corresponding `UndoLog` (§4.9.2) that can roll back the change when a transaction is aborted. In this case, the insert pushes the records onto to the existing vectors; the undo log discards the last item from each vector:

```
struct InsertPair<L,R>(L,R);

impl<L,R> RevertableOp<SplitVec<L,R>> for InsertPair<L,R> {
    type Err = std::convert::Infallible;
    type Log = UndoInsert;
    fn apply(self, rel: &mut SplitVec<L,R>)->Result<Self::Log,
Self::Err> {
        rel.left.push(self.0);
        rel.right.push(self.1);
        Ok(UndoInsert)
    }
}

struct UndoInsert;

impl<L,R> UndoLog<SplitVec<L,R>> for UndoInsert {
    fn revert(self, rel: &mut SplitVec<L,R>) {
        rel.left.pop();
        rel.right.pop();
    }
}
```

The `Insert` trait (§4.9.4) describes how to construct this operation for an arbitrary record. Here, it uses the `FromRecord` trait (§4.5.5) to split the inserted record into constituent parts:

```
impl<L,R,H> Insert<H> for SplitVec<L,R> where
    SplitVec<L,R>: RelationImpl,
    H: Header,
    L: FromRecord<H>,
    L::Remainder: Header + Record<Cols = L::Remainder>,
    R: FromRecord<L::Remainder>
{
    type Op = InsertPair<L,R>;
    type Remainder = <R as FromRecord<L::Remainder>>::Remainder;
    fn insert_op<Rec>(rec: Rec)->(Self::Op, Self::Remainder)
    where Rec: Record<Cols=H>
    {
        let (l, remainder) = L::from_rec(rec);
        let (r, remainder) = R::from_rec(remainder);
        (InsertPair(l,r), remainder)
    }
}
```

Deletion works similarly to insertion: The `Delete` trait (§4.9.5) describes how to generate a `RevertableOp` that will delete the specified records. Instead of a single record, however, `Delete` takes a `QueryFilter` (§4.6.2); all records that match the filter should be removed. While `Insert`'s undo log didn't contain any data, the undo log for `Delete` contains the removed records so that they can be restored if the transaction is aborted.

```
impl<L,R,Q> Delete<Q> for SplitVec<L,R>
where
    Q:QueryFilter,
    for<'a> (&'a L, &'a R):Record
{
    type Op = DeleteWhere<Q>;

    fn delete_op(q:Q)->Self::Op {
        DeleteWhere(q)
    }
}

pub struct DeleteWhere<Q>(Q);
```

(Continued)

```
impl<L,R,Q> RevertableOp<SplitVec<L,R>> for DeleteWhere<Q>
where
    for<'a> (&'a L, &'a R): Record,
    Q: QueryFilter,
{
    type Err = std::convert::Infallible;
    type Log = Removed<L,R>;

    fn apply(self, rel: &mut SplitVec<L,R>)
    ->Result<Self::Log, Self::Err> {
        let mut removed: Vec<(usize, L, R)> = vec![];
        for i in (0..rel.left.len()).rev() {
            if self.0.test_record(&(&rel.left[i], &rel.right[i]))
{
                removed.push(
                    (i, rel.left.remove(i), rel.right.remove(i))
                );
            }
        }
        Ok(Removed(removed))
    }
}

pub struct Removed<L,R>(Vec<(usize, L, R)>);

impl<L,R> UndoLog<SplitVec<L,R>> for Removed<L,R> {
    fn revert(mut self, rel: &mut SplitVec<L,R>) {
        for (i,l,r) in self.0.into_iter().rev() {
            rel.left.insert(i,l);
            rel.right.insert(i,r);
        }
    }
}
```

4.2 Relations

Relational algebra is primarily concerned with manipulating collections of records, known as *relations*. It defines a number of operators that both consume and produce these relations. A database is defined as a set of relations that contain all of its stored information. Conceptually, the process of retrieving particular information from the database involves using relational operators to construct a new relation, a *view*, which contains only the desired data, and then iterating over all the records in the view.

Memquery follows this model quite closely. It defines a number of storage types, with each making different tradeoffs between storage efficiency, modification performance, and query performance. It additionally defines a number of *view adapters*. Each of these adapters represents one of Codd's relational operators. These storage and adapter types all implement `Relation<'a>`, which is the main interface between Memquery and user code. It is also possible for library authors to define additional relations that will integrate seamlessly into the Memquery system.

4.2.1 Relation Trait

`Relation<'a>` is the main entry point for user code to interact with relations. It provides methods to iterate over all records as well as methods to construct restricted views into the relation. Many relations also support mutation via the `Insert` (§4.9.4) and `Delete` (§4.9.5) traits.

Application programmers should be familiar with the methods this trait provides. Library authors can implement `Relation` for additional types by providing appropriate `RelationImpl`, `QueryOutput`, and `IntoIterator` implementations as specified in the prerequisites (cf. §4.1.4).

Prerequisites

- `Self` must implement `RelationImpl` and `QueryOutput<'a>`
- `&'a Self` must implement `IntoIterator`. The resulting iterator must visit all the records contained in `Self`, and they must be of the type `Self::QueryOutput::QueryRow`.

`Relation` is automatically implemented for all types that meet these prerequisites.

Trait Parameters

`'a`: The lifetime during which returned results will remain valid.

Notable Implementors

- `Vec<`*T*`>` : Unordered storage of zero or more records of type *T*
- `Option<`*T*`>`: Storage of zero or one record of type *T*
- `BTreeIndex<`*K,T*`>`: Storage for zero or more records of type *T*, indexed by column *K* (§4.8.1)
- `RedundantIndex<`*K2,K1,Rel*`>`: A secondary index on column *K2* of the records in *Rel* (§4.8.2)
- *View Adapters*: Views into a relation, as described in §4.7

Methods

`iter_as<O>(&'a Self)->impl Iterator<Item=O>`

Projects each record in `Self` onto the type *O*. *O* must implement `FromExternalRecord<'a>` (§4.5.7), and produces a compile error if the columns in *O* are not a subset of the columns in `Self`.

```
col!{ProjectId: usize}
col!{QtyCommitted: usize}

let rel = vec![
    (ProjectId(1), QtyCommitted(3)),
    (ProjectId(2), QtyCommitted(5)),
    (ProjectId(3), QtyCommitted(7)),
];

let mut total = 0;
for &QtyCommitted(qty) in rel.iter_as() {
    total += qty;
}
total
⟹ 15
```

`truncate(&mut Self)->Result<(), impl Error>`

Atomically deletes all records in `Self`. Produces a compile error if `Self` does not implement `Delete` (§4.9.5). If any of the records cannot be deleted, no records are removed and `Result::Err(...)` is returned.

Truncate can be used in combination with `where_eq` to remove a subset of records:

```
col!{ProjectId: usize}
col!{QtyCommitted: usize}

let mut rel = vec![
    (ProjectId(1), QtyCommitted(3)),
    (ProjectId(2), QtyCommitted(5)),
    (ProjectId(3), QtyCommitted(7)),
];

assert!(rel.by_mut().where_eq(ProjectId(2)).truncate().is_ok());
rel
⟹ vec![
    (ProjectId(1), QtyCommitted(3)),
    (ProjectId(3), QtyCommitted(7)),
];
```

`by_ref(&Self)->RelProxy<&Self>`

Transforms a shared reference to `Self` into an object that implements `Relation`. This is primarily used to construct a view without transferring ownership of `Self` to the view.

`by_mut(&mut Self)->RelProxy<&mut Self>`

Transforms an exclusive reference to `Self` into an object that implements `Relation`. If `Self` implements `Insert` or `Delete` (§4.9.5), the returned object also implements them. This allows the construction of views that can modify the original relation.

`project<`*H*`>(Self)->ProjectedRel<Self,`*H*`>`

Constructs a projection view (§4.7.3) of `Self` that only allows access to the columns specified in *H*, which must be a `Header` (§4.4.1). Produces a compile error if *H* is not a subset of `Self::RelationImpl::Cols`.

`where_eq<`*C*`>(Self, `*C*`)->FilterRel<Self, Exact<`*C*`>>`

Constructs a filtered view (§4.7.2) of `Self` that contains those records which match the provided column value *C*. Produces a compile error if any of the following conditions are not met:

- *C* must implement `ColProxy` (§4.3.4)
- *C*`::For` must implement `PartialEq`
- `Self::RelationImpl::Cols` must contain *C*`::For`

```
col!{ProjectId: usize}
col!{QtyCommitted: usize}

let rel = vec![
    (ProjectId(1), QtyCommitted(3)),
    (ProjectId(2), QtyCommitted(7)),
    (ProjectId(3), QtyCommitted(5)),
];

rel.where_eq(ProjectId(2))
   .iter_as::<QtyCommitted>()
   .collect::<Vec<_>>()
⟹ vec![ QtyCommitted(7) ]
```

`where_in<C,R>(Self, R)->FilterRel<Self, ColRange<R>>`

Constructs a filtered view (§4.7.2) of `Self` that contains those records which fall within the specified range *R*. Produces a compile error if any of the following conditions are not met:

- *R* must implement `RangeBounds<C>`
- *C* must implement `ColProxy` (§4.3.4)
- *C*`::For` must implement `Ord`
- `Self::RelationImpl::Cols` must contain `C::For`

```
col!{ProjectId: usize}
col!{QtyCommitted: usize}

let rel = vec![
    (ProjectId(1), QtyCommitted(3)),
    (ProjectId(2), QtyCommitted(7)),
    (ProjectId(3), QtyCommitted(5)),
];

rel.where_in( QtyCommitted(5) .. )
   .iter_as::<ProjectId>()
   .collect::<Vec<_>>()
⟹ vec![ ProjectId(2), ProjectId(3) ]
```

`order_by<K>(Self, K)->OrderedRel<Self, K>`

Constructs a sorted view (§4.7.4) of `Self` that will produce results in the order specified by *K*, which must implement `SortKey` (§4.6.3).

```
col!{ProjectId: usize}
col!{QtyCommitted: usize}

let rel = vec![
    (ProjectId(1), QtyCommitted(3)),
    (ProjectId(2), QtyCommitted(7)),
    (ProjectId(3), QtyCommitted(5)),
];

rel.order_by::<sexpr!{Desc<QtyCommitted>}>()
   .iter_as::<ProjectId>()
   .collect::<Vec<_>>()
⟹ vec![ ProjectId(2), ProjectId(1), ProjectId(3) ]
```

`join<R>(Self, R) -> PeerJoin<Self,R>`

Constructs a peer join view (§4.7.6) between `Self` and *R*. Produces a compile error unless all of the following conditions are met:

- *R* implements `RelationImpl`
- The intersection between `Self`'s header and *R*'s header contains only one column, *C*
- *C* implements the `Eq` and `Col` traits

```
col!{ VendorId: usize }
col!{ VendorName: &'static str }
col!{ PartId: usize }

let parts = vec![
    (PartId(1), VendorId(1)),
    (PartId(2), VendorId(1)),
    (PartId(3), VendorId(2)),
]

let vendors = vec![
    (VendorId(1), VendorName("Acme, Inc.")),
    (VendorId(2), VendorName("FrobozzCo Intl."))
];

parts.join(vendors)
     .iter_as::<(PartId, VendorName)>()
     .collect::<Vec<_>>()

⟹ vec![
    (PartId(1), VendorName("Acme, Inc.")),
    (PartId(2), VendorName("Acme, Inc.")),
    (PartId(3), VendorName("FrobozzCo Intl.")),
]
```

`subjoin<C>(Self) -> SubordinateJoin<Self, C>`

Constucts a subordinate join view (§4.7.5) on the column *C*. Produces a compile error unless the following conditions are met:

- *C* implements Col
- `C::Inner` implements `RelationImpl`
- The headers of `Self` and `C::Inner` are disjoint

```
col!{ PartId: usize }
col!{ ProjectId: usize }
col!{ Parts: Vec<PartId> }

let projects = vec![
    (ProjectId(1), Parts(vec![])),
    (ProjectId(2), Parts(vec![PartId(1), PartId(2)])),
    (ProjectId(3), Parts(vec![PartId(3)])),
];

projects.subjoin::<Parts>()
        .iter_as::<(ProjectId, PartId)>()
        .collect::<Vec<_>>()
⟹ vec![
    (ProjectId(2), PartId(1)),
    (ProjectId(2), PartId(2)),
    (ProjectId(3), PartId(3)),
]
```

4.2.2 Implementing Relations

The most critical part of defining a new relation type is specifying how it will go about serving any particular query. Figure 4.1 shows an overview of the types and traits involved in specifying a new relation type *Rel*. In addition to defining the columns and record type that the relation holds, it is necessary to define one or more query plan types, *Plan*, that are responsible for executing queries against the relation. The Tylisp function `RelationImpl::Planner` is responsible for choosing which of these query plans should be used for any given `QueryRequest` (§4.6.1) *Req*.

Memquery provides `FallbackPlanner`, a generic query planner that will produce correct results for all queries. It uses a sequential-scan approach, and only requires a correct implementation of `IntoIterator` for `&Rel`. This fallback planner can be either used directly or as a last-resort planner to serve queries that the relation has not been specifically written to handle. It also provides two query plan implementations that are generic over all Relations: `FallbackPlan` will perform a sequential scan, and `PostSortPlan` will properly sort unordered results.

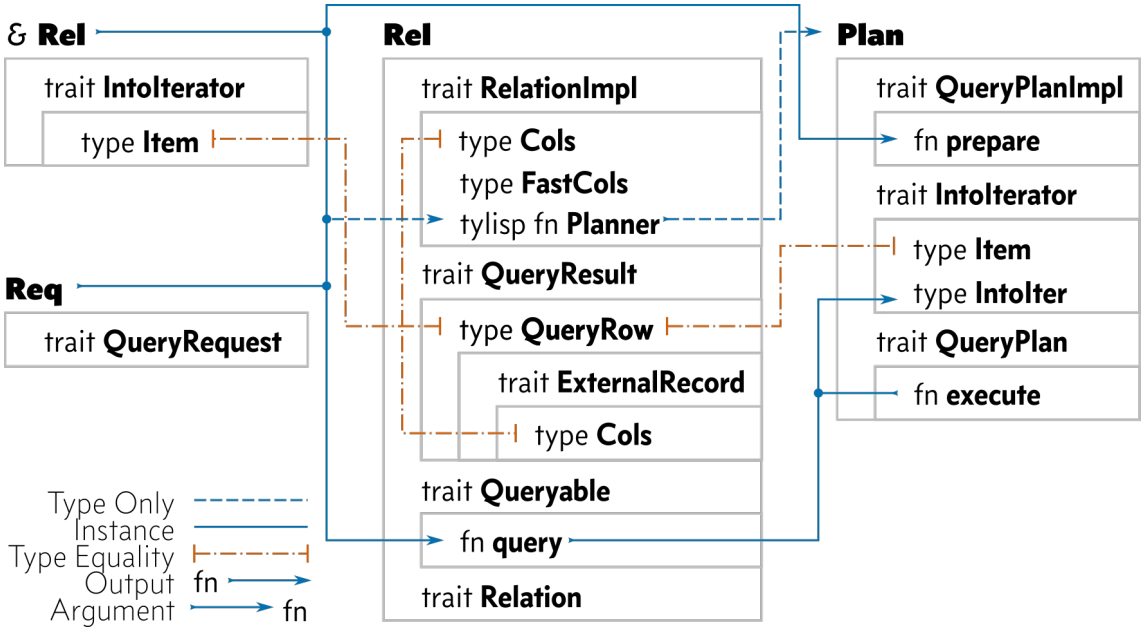

Figure 4.1: Relation and related traits (abridged)

4.2.3 RelationImpl Trait

`RelationImpl` indicates that `Self` is a type that should be treated as a relation. This trait serves as the entry point for Memquery to discover all necessary related types.

`RelationImpl` is mostly an implementation detail for the Relation trait. Application programmers need only be familiar with the columns stored in any given relation, represented in the `Cols` associated type.

Architects should additionally be familiar with the query plan and `FastCols` that each relation type uses, as these have a direct effect on program performance. Replacing one relation type with another that has the same header should not affect the correctness of query code.[8]

Library authors will need to correctly implement `RelationImpl` in order to integrate with the rest of the Memquery system. This involves implementing several traits on a variety of related types; see §4.1.4 for an overview of the entire process.

Prerequisites

None. Manually implemented.

Associated Types

- `Cols`: A `Header` which lists the columns contained in the relation's records.

8 To be a completely transparent replacement, the new relation type should have compatible Insert (§4.9.4) and Delete (§4.9.5) implementations as well.

- `FastCols`: A `Header` which indicates the columns that the relation can efficiently filter on. This is used solely as a hint for query planner, such as the join planner (§4.7.6).

- `Planner`: A Tylisp function that inspects a `QueryRequest` and returns a `QueryPlan` capable of fulfilling the request.

4.2.4 QueryOutput Trait

`QueryOutput<'a>` specifies the record type that will be returned from queries of `Self`. It cannot be merged into `RelationImpl` because of the presence of the lifetime parameter `'a`. The records returned from a query will include references into `Self`, and so must be generic over `'a`. On the other hand, the relational header of `Self`, for example, must be the same regardless of whether or not `Self` is currently being borrowed.

`QueryOutput` is primarily relevant to library authors that wish to define a relation type that composes with other, generic, relation types.

Application programmers should not rely on the particular `QueryRow` type specified by any relation, and instead use `Relation::iter_as()` or `Record::project_into()` to retrieve the column values required for any given computation. This leaves the architect free to replace the relation type with one that includes additional columns or uses a different record format.

Prerequisites

`Self` must implement `RelationImpl`.

Trait Parameters

`'a`: The lifetime during which returned results must remain valid.

Associated Types

`QueryRow`: The type of record produced by queries performed on `&'a Self`. This type must implement `ExternalRecord<'a>` and `QueryRow::Cols` must be identical to `Self::RelationImpl::Cols`.

4.2.5 Queryable Trait

`Queryable<'a, Req>` is the main entry point for evaluating a query. It provides the `query` method which will iterate over matching records in the order requested.

All relation types should implement `Queryable<'a, Q>` for every type Q which implements `QueryRequest`. Instead of implementing `Queryable` directly, library authors must ensure that `RelationImpl::Planner` always returns a compatible `QueryPlan` (§4.2.7) type, such as `FallbackPlan` (§4.2.9).

Application programmers should prefer constructing view adapters (§4.7) via the `Relation` trait (§4.2.1) to calling `query()` directly. For more advanced queries, this may not be possible. In this case, the programmer will need to become familiar with Memquery's query specification system (§4.6) in order to provide an appropriate parameter to `query()`.

Prerequisites

`Self` must implement both `RelationImpl` and `QueryOutput<'a>`. The type returned from calling `Self::Planner` must implement `QueryPlan<'a, Self, Req>`.

`Queryable` is automatically implemented for all relation types which satisfy these prerequisites. Properly implemented relations should implement `Queryable<'a, Q>` for any type `Q` which implements `QueryRequest`, but this is not enforced by the compiler.

Trait Parameters

- `'a`: The lifetime during which `Self` is borrowed for the query. This must include all regions in which one of the produced records exists.
- *Req*: A `QueryRequest` (§4.6.1) that specifies which records should be included in the query result, and the order in which they should be produced.

Associated Types

`Plan`: The type returned by `RelationImpl::Planner` for this request. Implements `QueryPlan<'a, Self, Req>`.

Methods

`query(&'a Self, Req)-><Self::Plan as IntoIterator>::IntoIter`

> Returns an `Iterator` over the query results. All of the returned records implement `ExternalRecord<'a, Cols=Self::Cols>`.

```
col!{A: usize}
col!{B: usize}

let rel = vec![(A(1), B(5)), (A(2), B(3)), (A(3), B(7))];
let mut result: Vec<(A,B)> = vec![];
for rec in rel.query( BlankRequest.set_order::<Desc<B>>() ) {
    result.push(rec.project());
}
result
⟹ vec![ (A(3), B(7)), (A(1), B(5)), (A(1), B(5)) ]
```

4.2.6 QueryPlanImpl Trait

`QueryPlanImpl<'a,Rel,Req>` indicates that `Self` is a concrete query plan to retrieve matching records from *Rel*. This trait provides a constructor, `prepare`. The query results should be provided by a suitable `IntoIterator` implementation.

The `QueryPlanImpl` trait is only relevant to library authors, who must implement it for any custom query plan. Code that interacts with other query plans should bound on the `QueryPlan` trait instead, which ensures that the plan can be executed after it has been constructed.

Prerequisites

None. Manually implemented.

Trait Parameters

- `'a`: The lifetime during which the query results will remain valid.
- *Rel*: The relation type being queried. Must implement `Relation<'a>` (§4.2.1)
- *Req*: A specification of the records to be retrieved. Must implement `QueryRequest` (§4.6.1)

Notable Implementors

`FallbackPlanner` (§4.2.8) is capable of executing any query request against any relation.

Associated Functions

`prepare(&'a Rel, Req)->Self`

Create an instance of `Self` to retrieve records from `Rel`.

4.2.7 QueryPlan Trait

`QueryPlan<'a,Rel,Req>` indicates that `Self` implements all traits necessary for executing the query *Req* against the relation *Rel*.

The `QueryPlan` trait is only relevant to library authors. Instead of implementing `QueryPlan` directly, library authors must implement `QueryPlanImpl` and `IntoIterator` for their custom query plan types. If these two implementations agree, Memquery will automatically implement the `QueryPlan` trait.

Prerequisites

`Self` must implement both `QueryPlanImpl<'a,`*Rel*`,`*Req*`>` and `IntoIterator`. The items yielded from the iterator must be of type `<`*Rel* `as QueryOutput<'a>>::QueryRow`.

`QueryPlan` is automatically implemented for all types that meet these prerequisites.

Trait Parameters

- `'a`: The lifetime during which the query results will remain valid.
- *Rel*: The relation type being queried. Must implement `Relation<'a>` (§4.2.1)
- `Req`: A specification of the records to be retrieved. Must implement `QueryRequest` (§4.6.1)

Notable Implementors

`FallbackPlanner` (§4.2.8) is capable of executing any query request against any relation.

Methods

`execute(Self)->><Self as IntoIterator>::IntoIter`

Produces an iterator of the query results

```
col!{ A: usize }
col!{ B: usize }

let rel = vec![(A(1), B(5)), (A(2), B(3)), (A(3), B(7))];
let mut result: Vec<(A,B)> = vec![];

let plan = FallbackPlan::prepare(
    &rel,
    BlankRequest.set_order::<Desc<B>>()
);

for rec in plan.execute() {
    result.push(rec.project());
}

result
⟹ vec![ (A(3), B(7)), (A(1), B(5)), (A(1), B(5)) ]
```

4.2.8 FallbackPlanner Type

`FallbackPlanner` is an ordinary Tylisp function which can be used as the query planner for any Relation. It uses a sequential scan to produce correct results for all query requests.

`FallbackPlanner` is only relevant to library authors. If a sequential scan is always the most appropriate query plan for a custom relation, its author can specify `RelationImpl::Planner = FallbackPlanner`, which avoids the need to write a custom query plan.

Syntax

`{FallbackPlanner, @&'a` R`, @`Q`} ⟹ FallbackPlan<'a,` R`,` Q`>`

> Returns a query plan type that uses a sequential-scan approach to retrieve results for the query Q from relation R.

Preconditions

R must implement `Relation<'a>` (§4.2.1), and Q must implement `QueryRequest` (§4.6.1).

4.2.9 FallbackPlan Type

`FallbackPlan<'a,` R`,` Q`>` is a sequential-scan query plan to execute the query Q against the relation R. It is a valid query plan for any combination of types R and Q where R implements `Relation<'a>` (§4.2.1) and Q implements `QueryRequest` (§4.6.1).

Library authors can use `FallbackPlan` to ensure that a relation's custom query planner can properly serve all possible queries: In the event that the user provides an unexpected query, the planner can return FallbackPlan instead of the relation's custom query plan.

```
defun!{ CanUseIndex { ... }}

defun!{ MyPlanner {
    ('a, Rel, Req) { &'a Rel, Req } =>
    {If, {CanUseIndex, @Req}, @MyIndexPlan<'a, Rel, Req>
                            , @FallbackPlan<'a, Rel, Req>};
}}
```

Type Parameters

`'a` is the program region where the query results will be valid

R is the relation type to be queried. Must implement `Relation<'a>` (§4.2.1)

Q is the query to be executed. Must implement `QueryRequest` (§4.6.1)

Query Plan

- Iterate over all records in *R*
- Discard records which do not match the selection filter *Q*`::Filters`
- If a particular sort order was requested:
  - Collect the results in a vector
  - Sort the vector according to *Q*`::OrderBy`

Implemented Traits

Table 4.1: Implemented Traits for FallbackPlan<'a,R,Q>

Trait	Bounds	Associated Types
`QueryPlanImpl<'a,`*R*`,`*Q*`>`	*R*`: Relation<'a>` *Q*`: QueryRequest`	
`IntoIterator`	*R*`: Relation<'a>` *Q*`: QueryRequest`	`Item =` *R*`::QueryRow`

4.2.10 PostSortPlan Type

`PostSortPlan<'a,` *R*`,` *Q*`>` is a query plan which will perform an unordered query against *R*, and then sort the results according to *Q*`::OrderBy`.

Library authors can use `PostSortPlan` to re-order query results according to the presentation order specified by *Q*:

```
defun!{ MyPlanner {
    ('a, Rel, Req:QueryRequest) {&'a Rel, Req} =>
        {If, {EmptyP, @Req::SortKey},
             @MyPlan<'a, Rel, Req::Filters>,
             @PostSortPlan<'a, Rel, Req>};
}}
```

Type Parameters

`'a` is the program region where the query results will be valid

R is the relation type to be queried. Must implement `Queryable<'a,` `ReplaceOrder<`*Q*`, HNil>>` (§4.2.5, 4.6.7)

Q is the query to be executed. Must implement `QueryRequest` (§4.6.1)

Requirements

R must use a different query plan for producing unordered results. Failure to do so may result in either a compile error or infinite recursion.

Query Plan

- Ask *R* to return records which satisfy the selection filter `Q::Filters`, and collect them into a vector.
- Sort the vector according to the defined presentation order `Q::OrderBy`.

Implemented Traits

Table 4.2: Implemented Traits for PostSortPlan<'a,R,Q>

Trait	Bounds	Associated Types
`QueryPlanImpl<'a,R,Q>`	`R: Relation<'a>` `Q: QueryRequest`	
`IntoIterator`	`R: Queryable<'a, ReplaceOrder<Q, HNil>>`	`Item = R::QueryRow`

4.3 Column Declarations

In relational algebra, columns are specified with two properties: The *domain* of valid values and a *role name* which specifies the semantic meaning of those values. In Memquery, each role is declared via the provided `col!` macro. For example, this defines a new public role, `PartId`, over the domain of 32-bit unsigned integers:

```
col!{ pub PartId: u32 }
```

A column's domain may represented by any Rust type that satifies the `Sized`, `Clone`, and `'static` bounds:

- `Sized` indicates that the data has a fixed size known as compile time. This ensures that the it can be stored directly inside records without any indirection or heap allocation.
- `Clone` allows values to be duplicated. This ensures that a redundant copy of the data can be stored in an index.
- `'static` prevents values from containing references that could expire. This is a technical limitation of Rust's runtime type reflection system (§2.2.5), which is used several places inside Memquery.

Values that don't satisfy `Sized` or `Clone` may be stored inside a column by wrapping them in one of Rust's standard containers: For example, a reference-counted pointer, `Rc`, implements both `Sized` and `Clone` even when its target doesn't.

The `'static` requirement can't be easily worked around in the general case. Most borrow types (those that carry a lifetime annotation), however, have owned counterparts that satisfy a `'static` bound. It is also possible to store raw pointers inside a column, but it is then the user's responsibility to ensure that they remain valid until they are no longer needed. Note that this bound requires the *type* to be valid for the entire program, but not necessarily any instances of that type.

4.3.1 Internal Representation

Figure 4.2 shows an overview of the relationships involved in the representation of a role type *C* with domain *T*. This structure is created by the user defining T as a normal Rust type and then invoking the macro `col!{ C : T }`, which defines both *C* and its related trait implementations. *C* is guaranteed to have the same memory representation as *T*, which allows references to *T* to be converted into references to *C* freely.

As this operation would normally be unsafe, the `Col` trait provides safe functions to perform this conversion, `wrap_ref` and `wrap_mut`. Similarly, it is sometimes necessary to change the role name of a value. This facility is provided by `Col`'s `rename*` methods, which will transform the value into any role of the same domain.

Many operations need to know the value of a particular column, but will work equally well with either owned or borrowed data. The `ColProxy` trait provides this abstraction, allowing generic functions and datatypes to accept owned and borrowed column data interchangeably.

Several of the implemented traits are primarily used by other parts of Memquery. The `LispId` and `Eval` traits (§3.5.1, 3.2.3) enable the role type to be processed by Tylisp. This, in turn, allows the `Header` trait (§4.4.1) to reject duplicate columns. The `Record` trait (§4.5.1) allows an individual data point to act as a single-column record.

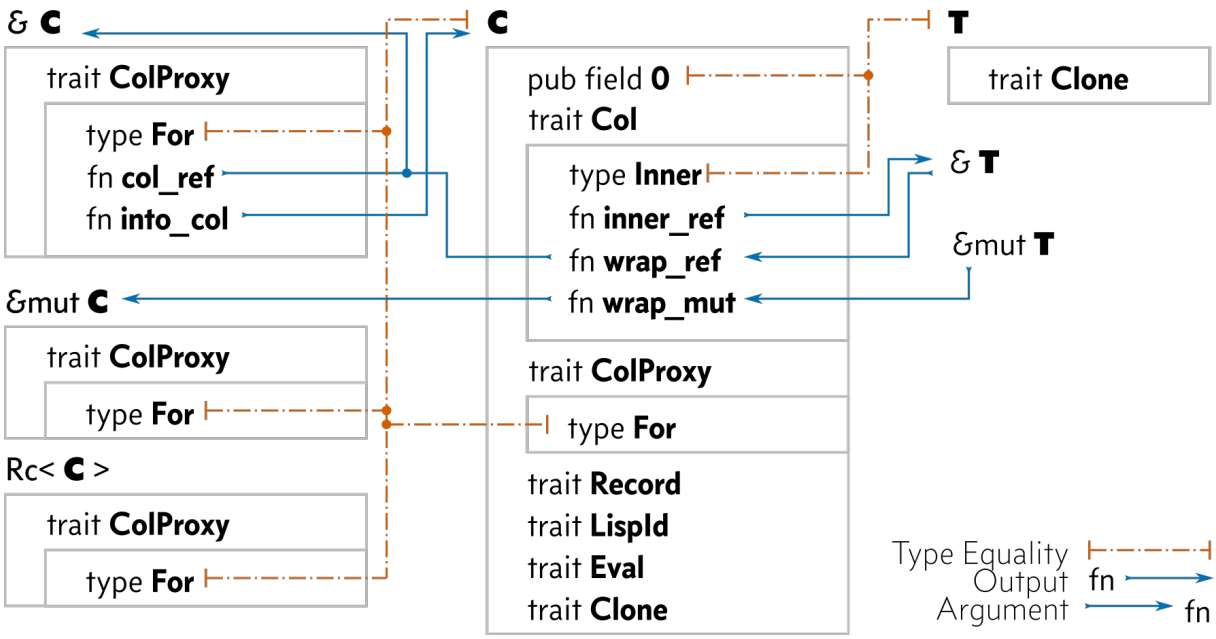

Figure 4.2: Col trait and related types (abridged)

4.3.2 col! Macro

The `col!` macro defines a role type that can be manipulated by Memquery records and relations.

Syntax

```
col!{ vis name : inner };
```

Parameters

- *vis* – An optional visibility specifier, such as `pub`, that will be applied to the generated struct
- *name* – An identifier that will be the name of the role type to be defined
- *inner* – A type that represents the allowable domain of values for the new role

Preconditions

The *name* must not conflict with any other type names in scope where the macro is called, and *inner* must conform to `Sized`, `Clone` and `'static` type bounds.

Macro Expansion

The macro defines a type called *name* in the current scope and provides a number of unconditional trait implementations:

- On *name*: `Col<Inner=`*inner*`>` (§4.3.3), `LispId` (§3.5.1), `Eval` (§3.2.3), `From<`*inner*`>`, `Into<`*inner*`>`, `Borrow<`*inner*`>`, `BorrowMut<`*inner*`>`, `Deref<Target=`*inner*`>`, `DerefMut`, `AsRef<`*inner*`>`, `AsMut<`*inner*`>`, `ColProxy<For=`*name*`>` (§4.3.4), `Record<Cols=sexpr!{` *name* `}>` (§4.5.1),

`FromRecordImpl<Cols=sexpr!{` *name* `}>` (§4.5.6),
`FromExternalRecord<'a, Cols=sexpr!{` *name* `}>` (§4.5.7)

- On `&'a` *name*: `From<&'a` *inner*`>`, `FromExternalRecord<&'a` *inner*`, Cols=sexpr!{`*name*`}>`

Additionally, if *inner* implements any of the traits `Debug`, `Copy`, `Hash`, `Default`, `Ord`, `PartialOrd`, `Eq`, or `PartialEq`, the corresponding trait is also implemented for *name*.

4.3.3 Col Trait

The `Col` trait indicates that `Self` is an individual data point managed by Memquery. It is may be used as an item inside a `Header` (§4.4.1). In the examples below, `PartId` and `AltPartId` are column types defined as following:

```
col!{ pub PartId: usize };
col!{ pub AltPartId: usize };
```

Prerequisites

Types that implement the `Col` trait must be defined via the `col!` macro. Refer to §4.3.2 for details.

Associated Types

`Inner`: The domain of values that can be stored within this column. Must satisfy a `'static` bound and implement the traits `Sized` and `Clone`.

Methods

`inner_ref(&self)->&Self::Inner`

Returns a reference to the stored domain value.

```
PartId(6).inner_ref() ⟹ &6
```

`rename<`*C*`>(self)->`*C*

Constructs an instance of the column type *C* with the same domain value as `Self`. Produces a compile error if *C* does not implement `Col` or *C*`::Inner` is not the same as `Self::Inner`.

```
PartId(6).rename::<AltPartId>() ⟹ AltPartId(6)
```

`rename_ref<`*C*`>(&self)->&`*C*

Renames the column type of a shared reference, without constructing a new instance. Produces a compile error if *C* does not implement `Col` or *C*`::Inner` is not the same as `Self::Inner`.

```
PartId(6).rename_ref::<AltPartId>() ⟹ &AltPartId(6)
```

`rename_mut<C>(&mut self)->&mut C`

Renames the column type of an exclusive reference, without constructing a new instance. Produces a compile error if *C* does not implement `Col` or *C*`::Inner` is not the same as `Self::Inner`.

```
let mut part_id = PartId(6);
let alt_part_id = part_id.rename_mut::<AltPartId>();
*alt_part_id = AltPartId(42);
part_id
⟹ &PartId(42)
```

Associated Functions

`wrap_ref(&Self::Inner)->&Self`

Renames a shared reference to the domain type as a shared reference to Self.

```
PartId::wrap_ref(&7) ⟹ &PartId(7)
```

`wrap_mut(&mut Self::Inner)->&mut Self`

Renames an exclusive reference to the domain type as an exclusive reference to `Self`.

```
let mut id = 5;
let part_id = PartId::wrap_mut(&id)
*part_id = PartId(7);
id
⟹ 7
```

4.3.4 ColProxy Trait

Indicates that `Self` provides access to a single Memquery data point, possibly by reference.

Prerequisites

None. For every type *C*`:Col`, `ColProxy` is automatically implemented for *C*, &*C*, `&mut` *C*, `Box<`*C*`>`, `Rc<`*C*`>`, and `Arc<`*C*`>`. Users may implement it manually for custom types.

Associated Types

`For`: The column type that `Self` provides access to. Must implement `Col`

Methods

`into_col(self)->Self::For`

Transforms `Self` into its proxied column type. For reference types, this clones the value.

```
(&PartId(5)).into_col() ⟹ PartId(5)
AltPartId(42).into_col() ⟹ AltPartId(42)
```

`col_ref(&self)->&Self::For`

Retrieves a shared reference to the proxied column type

```
(&PartId(5)).col_ref() ⟹ &PartId(5)
AltPartId(42).col_ref() ⟹ &AltPartId(42)
```

4.4 Relational Headers

In relational algebra, operations, records, and relations are primarily characterized by a set of role names, a *header*. In Memquery, this is represented by an `HList` of role types which contains no duplicates. The `Header` trait is automatically implemented for all such lists.

Header types have a dual use. In most cases, they are never instantiated and simply represent properties of some other type, such as a record or relation. When instantiated, they serve as a lowest-common-denominator record type: Any record can be converted into its header, which can then be manipulated by `HList` operations.

Figure 4.3 shows an overview of some of the traits implemented for headers made up of columns *A, B, C, etc:*

- `Header` provides a suite of useful methods for user code,
- `HasCol<`*X*`>` allows code to be bounded on the presence of the column *X*,
- `ProjectFrom` divides a header into two parts, and
- `AsListRefs` transforms a reference to a header into a list of column references .

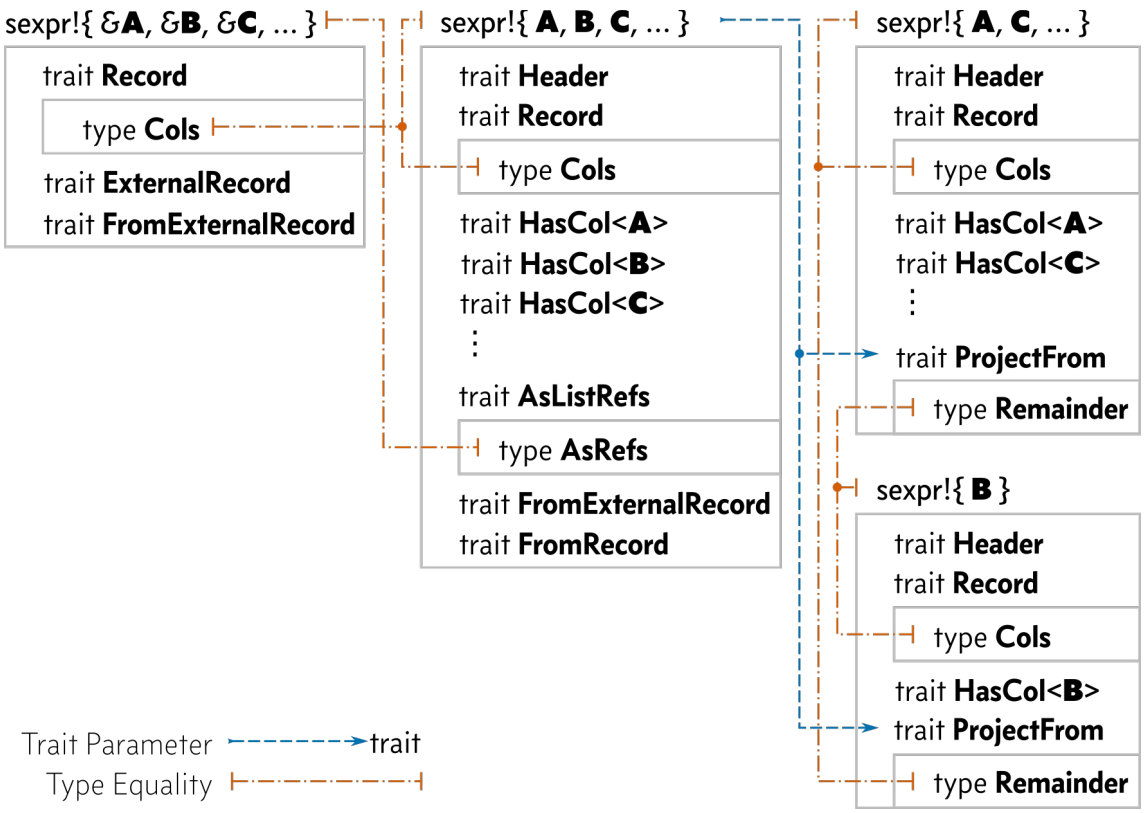

Figure 4.3: Header types and related traits (abridged)

4.4.1 Header Trait

`Header` indicates that `Self` is a set of column types, with no duplicates.

Prerequisites

`Self` must implement `List` (§3.1.3), and every element of the list must implement the `Col` trait (§4.3.3). No two elements may be of the same type.

`Header` is automatically implemented for all types that meet these prerequisites.

Associated Functions

`has_col<`*C*`>()->bool`

> Returns `true` if the column type *C* is a member of `Self`. C must implement the `Col` trait (§4.3.3).

```
col!{A:usize}
col!{B:usize}
col!{C:usize}
type H = sexpr!{A,B};

H::has_col::<C>() ⟹ false
H::has_col::<A>() ⟹ true
```

`is_disjoint<H>()->bool`

> Returns `true` if the type *H* contains no columns that are members of `Self`. *H* must implement `Header`.

```
col!{A:usize}
col!{B:usize}
col!{C:usize}
type H = sexpr!{A,B};

H::is_disjoint::<sexpr!{B,C}>() ⟹ false
H::is_disjoint::<sexpr!{C}>() ⟹ true
```

`clone_from_rec<R>(&R)->Self`

> Constructs an instance of `Self` by cloning the relevant columns from the record *R*. *R* must implement the `Record` trait (§4.5.1), and all columns in `Self` must be present in *R*`::Cols`.

`clone_from_rec_unchecked<R>(&R)->Self`

> Constructs an instance of `Self` by cloning the relevant columns from the record *R*. *R* must implement the `Record` trait (§4.5.1); produces a runtime panic if any member column of `Self` is absent from *R*`::Cols`.

Methods

`col_opt<C>(&Self)->Option<&C>`

> Returns a reference to the column *C* contained within `Self`. *C* must implement the `Col` trait (§4.3.3). If *C* is not an element of `Self`, returns `None`.

`col_ref<C>(&Self)->&C`

> Returns a reference to the column *C* contained within `Self`. *C* must implement the `Col` trait (§4.3.3) and be an element of `Self`.

4.4.2 HasCol Trait

`HasCol<C>` indicates that the column *C* is a member of `Self`

Prerequisites

`Self` must implement `Header`. *C* must implement the `Col` trait (§4.3.3) and be a member of `Self`.

`HasCol` is automatically implemented for all types that satisfy these prerequisites.

Trait Parameters

C is the column type whose presence is being tested. Must implement the `Col` trait (§4.3.3) and be a member of `Self`.

Associated Types

`Index` is a locator type, either `There<...>` or `Here` (§3.1.9), that represents the location of *C* within `Self`. `Index` is never `Missing`, as *C* is guaranteed to be present.

Table 4.3: Sample HasCol Implementations

`Self`	`HasCol<A>::Index`
`sexpr!{A,B,C}`	`Here`
`sexpr!{B,A,C}`	`There<Here>`
`sexpr!{B,C}`	*not implemented*

4.4.3 ProjectFrom Trait

`ProjectFrom<`*H*`>` constructs `Self` from an instance of *H*

Prerequisites

`Self` must implement `Header`, and the elements of `Self` must be a subset of the elements of *H*.

`ProjectFrom` is automatically implemented for all types which meet these prerequisites.

Type Parameters

H is the type which is the source of the projection data. Must implement the `Header` trait.

Associated Types

`Remainder` is a list of the elements in *H* that are not members of `Self`. Must implement the `Header` trait.

Table 4.4: Sample ProjectFrom Implementations

`Self`	***H***	`ProjectFrom<`***H***`>::Remainder`
`sexpr!{A,B,C}`	`sexpr!{A,B,C}`	`sexpr!{}`
`sexpr!{C,B,A}`	`sexpr!{A,B,C}`	`sexpr!{}`
`sexpr!{B}`	`sexpr!{A,B,C}`	`sexpr!{A,C}`
`sexpr!{A,B,C}`	`sexpr!{A,C}`	*not implemented*

Associated Fumctions

`fn project_from(`*H*`)->(Self, Self::Remainder)`

Constructs both `Self` and `Remainder` from *H*.

```
<sexpr!{A,B}>::project_from(sexpr_val!{B(2), A(1), C(3)})
⟹ (sexpr_val!{A(1), B(2)}, sexpr_val!{C(3)})

<sexpr!{C}>::project_from(sexpr_val!{B(2), A(1), C(3)})
⟹ (sexpr_val!{C(3)}, sexpr_val!{B(2), A(1)})
```

4.4.4 AsListRefs Trait

The `AsListRefs<'a>` trait provides a method to transform an `ExternalRecord<'a>` (§4.5.2) into a list of column references.

Trait Parameters

`'a` is the lifetime in which the references will be valid

Prerequisites

`Self` must be a `Header`. Automatically implemented for all types that implement `Header`.

Associated Types

`AsRefs` a list of column references. Implements `List`, and every element is of the form `&'a` *C*, where *C* is an element of `Self`.

Table 4.5: Sample AsListRefs<'a> Implementations

Self	AsListRefs<'a>::AsRefs
sexpr!{}	sexpr!{}
sexpr!{A,B}	sexpr!{&'a A, &'a B}

Methods

`ref_from_ext_rec<`*R*`>(&`*R*`)->Self::AsRefs`

Retrieve column references from the record *R*. *R* must implement `ExternalRecord<'a>` (§4.5.2)

```
<sexpr!{B,A}>::ref_from_ext_rec( &(A(5), B(7), C(42)) )
⟹ sexpr_val!{&B(7), &A(5) }
```

4.5 Records

Memquery represents relational tuples via the `Record` trait. Some records, such as those returned by queries, contained borrowed data. These implement the `ExternalRecord<'a>` trait, which provides methods to retrieve references which can outlive the record type itself. Table 4.6 summarizes the primitive types that implement these traits.

The `FromRecord`, `FromRecordImpl`, and `FromExternalRecord` traits provide a standardized interface for transforming record types.

Table 4.6: Record implementations for primitive types

Type	Record Bounds	`ExternalRecord<'a>` Bounds
C	C`: Col`	*Not implemented*
`&'a` C	C`: Col`	`Self: Record`
`sexpr!{`C_1`,` C_2`, ... }`	$\forall i.\ C_i$`: Col` $\forall i.\ \forall j.\ (i \neq j) \Rightarrow (C_i \neq C_j)$	*Not implemented*
`sexpr!{&'a` C_1`,` `&'a` C_2`, ...}`	$\forall i.\ C_i$`: Col` $\forall i.\ \forall j.\ (i \neq j) \Rightarrow (C_i \neq C_j)$	`Self: Record`
`&'a` R	R`: Record`	`Self: Record`
$(R_1, R_2, \ldots)$	$\forall i.\ R_i$`: Record` $\forall i.\ \forall j.\ (i \neq j) \Rightarrow$ R_i`::Cols` $\cap$ R_j`::Cols = ` $\emptyset$	`Self: Record` $\forall i.$ R_i`: ExternalRecord<'a>`

4.5.1 Record Trait

`Record` indicates that `Self` is a relational tuple.

Requirements

None.

Associated Types

`Cols` specifies the columns present in the record. Must implement `Header` (§4.4.1).

Methods

`into_cols(Self)->Self::Cols`

> Transforms `Self` into a `Header` instance. This can be used to unify the types of various records with a common header.

`clone_cols(&Self)->Cols`

Constructs a new `Header` instance without consuming Self.

`col_ref<C>(&Self)->&C`

Retrieve a reference to the column type *C*. *C* must implement `Col` (§4.3.3), and be a member of `Self::Cols`.

`col_opt<C>(&Self)->Option<C>`

Retrieve a reference to the column type *C*. Returns `None` if *C* is not a member of `Self::Cols`.

C must implement the `Col` trait (§4.3.3).

`project<H>(Self)->Projection<Self, H>`

Construct a view of this record which only allows access to columns which are members of *H*. *H* must implement the `Header` and `ProjectFrom<Self::Cols>` traits (§4.4.1, 4.4.3)

`project_into<R>(Self)->R`

Construct an instance of type *R* based on the data stored in `Self`. *R* must implement `FromRecord<Self::Cols>` (§4.5.5).

`rename_col<A,B>(Self)->Rename<Self,A,B>`

Construct a view of this record that replaces the column type *A* with *B*. *A* and B must both implement `Col` (§4.3.3), and *A*`::Inner` must be the same type as *B*`::Inner`.

Implementing Record

Defining a record type only requires specifying its header and providing one method implementation, `col_opt`. This is exactly analogous to `Header::col_opt` (§4.4.1) and must return `Some(...)` for any role listed in its header. Other access methods can also be overridden where a more performant approach is available than the default implementation.

A typical implementation is shown below. This example is taken from the inventory management case study described in §5. The `Commitment` structure describes the quantity of a particular part that has been assigned for use in the specified project.

Though the `col_opt` function here uses Rust's runtime dispatch system (§2.2.5), it is designed to be devirtualized by the optimizer. Because Rust monomorphizes generic functions, the code for each role type `C` will be generated separately. `TypeId::of` is a

constant function, so constant folding will reduce each condition to a simple `true` or `false`. Dead code elimination then leaves only one possible value for the vtable pointer stored inside the `dyn Any` object. The efficacy of these techniques has been confirmed by disassembling the output of similar functions.

```
col!{ pub PartId: usize }
col!{ pub ProjectId: usize }
col!{ pub QtyCommitted: usize }

pub struct Commitment {
    part_id: usize,
    project_id: usize,
    quantity_committed: usize
}

impl Record for Commitment {
    type Cols = sexpr!{ PartId, ProjectId, QtyCommitted };

    fn col_opt<C:Col>(&self)->Option<&C> {
        use std::any::{Any, TypeId};
        if TypeId::of::<C>() == TypeId::of::<PartId> {
            PartId::wrap_ref(&self.part_id) as &dyn Any
        } else if TypeId::of::<C>() == TypeId::of::<ProjectId> {
            ProjectId::wrap_ref(&self.project_id) as &dyn Any
        } else {
            QtyCommitted::wrap_ref(&self.project_id) as &dyn Any
        }.downcast_ref()
    }
}
```

4.5.2 ExternalRecord Trait

`ExternalRecord<'a>` indicates that `Self` is a record that contains data borrowed for the lifetime `'a`.

Requirements

`Self` must implement `Record`

Trait Parameters

`'a` is the program region in which the column data is known to remain valid.

Methods

`ext_col_ref<`*C*`>(&Self)->&'a` *C*

> Retrieve a reference to the column type *C*. *C* must implement the `Col` trait (§4.3.3) and be a member of `Self::Cols`.

`ext_col_opt<C>(&Self)->Option<&'a C>`

> Retrieve a reference to the column type *C*, which must implement the `Col` trait (§4.3.3). Returns `None` if *C* is not a member of `Self::Cols`.

4.5.3 Projection Type

`Projection<R, H>` is a record that represents data from type *R* for columns present in the header *H*.

Type Parameters

R is the type which contains the column data. Must implement the `Record` trait (§4.5.1).

H specifies the columns accessible via the projection. Must implement the `Header` and `ProjectFrom<R::Cols>` traits (§4.4.1, 4.4.3).

Construction

`Projection` instances are obtained via the `project` method of the `Record` trait.

Implemented Traits

All `Projection` instances implement `Record`. They also implement `ExternalRecord<'a>` when *R* implements `ExternalRecord<'a>` (§4.5.2).

4.5.4 Rename Type

`Rename<R,A,B>` is a record that represents the data from record *R* where column *A* has been renamed to *B*.

Type Parameters

R is the type which provides the underlying data. Must implement the `Record` trait.

A is the column that will be replaced. Must implement the `Col` trait (§4.3.3).

B is the column type that will replace *A*. Must implement the `Col` trait and be absent from *R*`::Cols`. *B*`::Inner` must be the same type as *A*`::Inner`.

Construction

`Rename` instances are obtained via the `rename_col` method of the `Record` trait (§4.5.1).

Implemented Traits

All `Rename` instances implement `Record`. They also implement `ExternalRecord<'a>` when *R* implements `ExternalRecord<'a>` (§4.5.2).

4.5.5 FromRecord Trait

`FromRecord<H>` indicates that `Self` can be constructed from records that contain the columns in *H*.

Trait Parameters

H is the relational header of the source record. Must implement the `Header` and `ProjectFrom<<Self as FromRecordImpl>::Cols>` traits (§4.4.1, 4.4.3).

Prerequisites

`Self` must implement the `FromRecordImpl` trait (§4.5.6). `FromRecord<H>` is automatically implemented for these types where *H* meets satisfies the requirements listed in "Trait Parameters".

Associated Types

`Remainder` contains the columns of *H* that are not consumed during the creation of `Self`. Implements the `Header` trait.

Associated Functions

`from_rec<R>(R)->(Self, Self::Remainder)`

> Construct an instance of `Self` using the data in *R*, which must implement `Record<Cols=H>`. Usually invoked via the `project_into` method of the `Record` trait (§4.5.1).

4.5.6 FromRecordImpl Trait

`FromRecordImpl` specifies the portion of the `FromRecord` implementation that is specific to the type `Self`.

Associated Types

`Cols` specifies the columns that will be consumed when constructing `Self`. Must implement `Header` (§4.4.1).

Associated Functions

`from_rec_raw<R>(R)->Self`

> Consumes *R* to construct `Self`. *R* must implement `Record<Cols=Self::Cols>` (§4.5.1).

Example Implementation

```
impl FromRecordImpl for Commitment {
    type Cols = sexpr!{ PartId, ProjectId, QtyCommitted };

    fn from_rec_raw(r: impl Record<Cols=Self::Cols>)->Self {
        let sexpr_pat!{part:_, proj:_, qty:_} = r.into_cols();
        Commit {
            part_id: *part,
            project_id: *proj,
            quantity_committed: *qty
        }
    }
}
```

4.5.7 FromExternalRecord Trait

`FromExternalRecord<'a>` provides a method to construct `Self` from a record containing borrowed data.

Trait Parameters

`'a` is the program region in which the borrowed data is known to be valid.

Associated Types

`Cols` is specifies the columns that must be present to construct `Self`. Must implement `Header` (§4.4.1).

Associated Functions

`from_ext_rec_raw<`*R*`>(`*R*`)->Self`

> Constructs `Self` from the borrowed data contained in *R*, which must implement `ExternalRecord<'a>` and `Record<Cols=Self::Cols>` (§4.5.1-2).

`from_ext_rec<`*R*`>(`*R*`)->Self`

> Constructs `Self` from the borrowed data contained in *R*, which must implement `ExternalRecord<'a>`. `Self::Cols` must implement `ProjectFrom<`*R*`::Cols>` (§4.4.3). Usually invoked via the `iter_as` method of the `Relation` trait (§4.2.1).

Example Implementation

There is a default implementation for `from_ext_rec`, so implementors need only write `from_ext_rec_raw`.

```
pub struct QueryResultRef<'a> {
    pub part_id: usize,
    pub part_name: &'a str,
    pub quantity_committed: usize,
}

impl<'a> FromExternalRecord<'a> for QueryResultRef<'a> {
    type Cols = sexpr!{PartId, PartName, QtyCommitted};
    fn from_ext_rec_raw(r: impl ExternalRecord<'a,
Cols=Self::Cols>)->Self {
        QueryResultRef {
            part_id: r.ext_col_ref::<PartId>().0,
            part_name: r.ext_col_ref::<PartName>().as_str(),
            quantity_committed: r.ext_col_ref::<QtyCommitted>().0,
        }
    }
}
```

4.6 Query Specification

Each query is represented by an object which implements the `QueryRequest` trait, which specifies a selection filter and presentation order. The selection filter is a conjunction of individual terms, each of which implement the `QueryFilter` trait. Each of these are represented as a list of individual terms, which may be of varying types.

Library authors need to be familiar with all of the traits and types described in this chapter. Application programmers may wish to learn this material as well in order to express queries that would be cumbersome or impossible with view adapters (§4.7). Architects do not need to use these directly, but understanding how queries are represented may aid their analysis of how different relations will perform when presented with particular queries.

4.6.1 QueryRequest Trait

`QueryRequest` indicates that `Self` is a query specification.

Requirements

Must implement `Clone`.

Associated Types

`Filters` is a list of filter terms; only records which satisfy all terms will be returned from this query. Must implement the `List` trait (§3.1.3), and every list element must implement the `QueryFilter` trait (§4.6.2).

`OrderBy` is list of sort keys. Records returned from this query will be presented in the lexicographic order defined by this list. Must implement the `List` trait, and every list element must implement the `SortKey` trait (§4.6.3).

Methods

`into_filters(Self)->Self::Filters`

> Retrieves an instance of the query's selection filters, which may contain captured variables.

`add_filter<F>(Self, F)->AddFilter<Self,F>`

> Constructs a query which selects records that match $F \wedge$ `Self::Filters`, and preserves presentation order. *F* must implement `QueryFilter`.

`set_filters<F>(Self, F)->ReplaceFilters<Self,F>`

> Constructs a query which selects records that match the filter list *F*, while preserving the presentation order. *F* must implement the List trait, and every element of *F* must implement `QueryFilter`.

`set_order<O>(Self)->ReplaceOrder<Self,O>`

> Constructs a query which selects records that match `Self::Filters`, presented in the order specified by *O*. *O* must implement the `List` trait, and every element of *O* must implement the `SortKey` trait.

4.6.2 QueryFilter Trait

`QueryFilter` indicates that `Self` is a record-selection operator.

Associated Types

`ReqCols` specifies the columns inspected by this filter. Must implement the `Header` trait (§4.4.1).

Methods

`test_record<R>(&self, &R)->bool`

> Check whether the given record of type *R* should be selected. If *R* is missing any columns from `Self::ReqCols`, this method must return `true` if there exists any combination of the missing values which would allow the record to be selected.

`bounds_of<`*C*`>(&self)->(std::ops::Bound<&`*C*`>, std::ops::Bound<&`*C*`>)`

> Returns a range of valid values for the column *C*. If a record *r* contains a *C* value that falls outside this range, `self.test_record(&`*r*`)` must return `false`. *C* must implement the `Col` and `Ord` traits (§4.3.3).

4.6.3 SortKey Trait

`SortKey` indicates that `Self` specifies a presentation order for records.

Associated Types

`ReqCols` specifies the columns which are inspected to determine the ordering. Must implement `Header` (§4.4.1).

Methods

`cmp<`*R*`>(a:&`*R*`, b:&`*R*`)->std::cmp::Ordering`

> Returns the relative ordering between records *a* and *b*. *R* must implement `Record` (§4.5.1), and *R*`::Cols` must be a superset of `Self::ReqCols`.

`sort<`*I*`,`*R*`>(`*I*`)->impl Iterator<Item=`*R*`>`

> Returns the records yielded from the iterator I in the specified presentation order. *I* must implement `Iterator<Item=`*R*`>`. *R* must implement `Record` (§4.5.1), and *R*`::Cols` must be a superset of `Self::ReqCols`.

4.6.4 BlankRequest Type

`BlankRequest` represents a query that returns all records in a relation, in an arbitrary order.

Implemented Traits

`QueryRequest< Filters = sexpr!{}, OrderBy = sexpr!{} >` (§4.6.1)

4.6.5 AddFilter Type

`AddFilter<`*Q*`,`*F*`>` represents a query that contains all the results from a query *Q* that also satisfy the filter *F*.

Type Parameters

Q is the source query. It must implement `QueryRequest` (§4.6.1)

F is the filter term to be added. Must implement `QueryFilter` (§4.6.2).

Construction

`AddFilter` instances are obtained via the `add_filter` method of the `QueryRequest` trait.

Implemented Traits

Table 4.7: Implemented traits for AddFilter<Q,F>

Trait	Bounds	Associated Types
`QueryRequest`	Q`: QueryRequest` F`: QueryFilter`	`Filters = sexpr!{`F`; `Q`::Filters}` `OrderBy = `Q`::OrderBy`

4.6.6 ReplaceFilters Type

`ReplaceFilters<`Q`,`F`>` represents a query that selects records according to F and presents them in the order specified by Q.

Type Parameters

Q is the query that determines the presentation order. Must implement `QueryRequest` (§4.6.1)

F is the new selection filter list. Must implement `List` and every element must implement `QueryFilter` (§3.1.3, 4.6.2).

Construction

`ReplaceFilter` instances are obtained via the `replace_filters` method of the `QueryRequest` trait.

Implemented Traits

Table 4.8: Implemented traits for ReplaceFilter<Q,F>

Trait	Bounds	Associated Types
`QueryRequest`	Q`: QueryRequest` F`: List` `∀(`I`∈`F`). `I`: QueryFilter`	`Filters = `F `OrderBy = `Q`::OrderBy`

4.6.7 ReplaceOrder Type

`ReplaceOrder<`Q`,`O`>` represents a query that returns the results specified by the query Q presented in the order O.

Type Parameters

Q is the source query. Must implement `QueryRequest` (§4.6.1)

O is the new presentation order. Must implement `List` and every element must implement `SortKey` (§3.1.3, 4.6.3).

Construction

`ReplaceOrder` instances are obtained via the `set_order` method of the `QueryRequest` trait.

Implemented Traits

Table 4.9: Implemented traits for ReplaceOrder<Q,O>

Trait	Bounds	Associated Types
`QueryRequest`	Q`: QueryRequest` O`: List` $\forall(I \in O).\ I$`: SortKey`	`Filters =` Q`::Filters` `OrderBy =` O

4.6.8 Exact Type

`Exact<`C`>` is a filter term which selects records with a specific value for column C.

Type Parameters

C specifies the column to be constrained. Must implement `ColProxy` (§4.3.4), and C`::For` must implement `Eq`.

Construction

`Exact` instances can be constructed directly:

```
col!{A: usize}
let a = A(42);
let exact = Exact(&a);
```

Implemented Traits

Table 4.10: Implemented traits for Exact<C>

Trait	Bounds	Associated Types
`QueryFilter`	C`: ColProxy`	`ReqCols: sexpr!{`C`::For}`
`Clone`	C`: Clone`	

4.6.9 ColRange

`ColRange<C>` is a filter term which selects records within a specific range for column *C*.

Type Parameters

C specifies the column to be constrained. Must implement `ColProxy` (§4.3.4), and *C*`::For` must implement `Ord`.

Construction

`ColRange` instances can be constructed via the `From` trait:

```
col!{A: usize}
let range = ColRange::from( A(7) .. A(42) );
```

Implemented Traits

Table 4.11: Implemented traits for ColRange<C>

Trait	Bounds	Associated Types
`QueryFilter`	*C*`: ColProxy`	`ReqCols: sexpr!{`*C*`::For}`
`From<`*B*`>`	*B*`: RangeBounds<`*C*`>`	
`Clone`	*C*`: Clone`	

4.6.10 Asc Type

`Asc<C>` is a presentation order that sorts records in ascending order of the column value *C*.

Type Parameters

C is the column to be inspected for this ordering. Must implement the `Col` (§4.3.3) and `Ord` traits.

Construction

The `Asc` type is only used for compile-time information; no instances are ever constructed.

Implemented Traits

Table 4.12: Implemented traits for Asc<C>

Trait	Bounds	Associated Types
`SortKey`	*C*`: Col + Ord`	`ReqCols: sexpr!{`*C*`}`

4.6.11 Desc Type

`Desc<C>` is a presentation order that sorts records in descending order of the column value *C*.

Type Parameters

C is the column to be inspected for this ordering. Must implement the `Col` (§4.3.3) and `Ord` traits.

Construction

The `Desc` type is only used for compile-time information; no instances are ever constructed.

Implemented Traits

Table 4.13: Implemented traits for Desc<C>

Trait	Bounds	Associated Types
`SortKey`	`C: Col + Ord`	`ReqCols: sexpr!{C}`

4.7 View Adapters

In Memquery, relational operators are represented by adapter objects which are themselves relations; each adapter object captures its input relations. This is analogous to the behavior of iterator adapters in Rust's standard library (§Error: Reference source not found). Because ownership of the source relations is transferred to the adapter object, the sources will be unusable after an adapter is created. As this is sometimes undesirable, Memquery provides a mechanism to transform references into relation types, `RelProxy`.

Application programmers should be familiar with the capabilities of all the types in this chapter, as they form the primary query interface for application code. Architects may also find them useful for constructing general-purpose views. Library authors will usually be working at a lower level, and will rarely find these types directly useful.

4.7.1 RelProxy Type

`RelProxy<P>` is an adapter type that allows a reference *P* to a relation to be used as a relation.

Type Parameters

P: A smart pointer type that refers to a relation. Must implement `Deref`, and *P*`::Target` must implement `RelationImpl` (§4.2.3)

Query Plan

All queries are passed unchanged to *P*`::Target`

Associated Functions

`new(`*P*`) -> RelProxy<`*P*`>`

> Constructs a new `RelProxy` object. `RelProxy` objects can also be created via the `Relation` trait (§4.2.1) methods `by_ref` and `by_mut`.

Implemented Traits

Table 4.14 lists the traits directly implemented for `RelProxy<`*P*`>`, where `P::Target =` *R*.

Table 4.14: Implemented Traits for RelProxy<P: Deref<Target=R>>

Trait	Additional Bounds	Associated Types
`Deref`		`Target =` *R*
`DerefMut`	*P*`: DerefMut`	`Target =` *R*
`RelationImpl`	*R*`: RelationImpl`	`Cols =` *R*`::Cols` `FastCols =` *R*`::FastCols`
`QueryOutput<'a>`	*R*`: QueryOutput<'a>`	`QueryRow =` *R*`::QueryRow`
`Insert<`*H*`>`	*P*`: DerefMut,` *R*`: Insert<`*H*`>`	
`Delete<`*Q*`>`	*P*`: DerefMut,` *R*`: Delete<`*Q*`>`	

4.7.2 FilterRel Type

`FilterRel<`*R*`,`*F*`>` is a relation adapter that represents the selection of records from *R* which match the filter *F*

Type Parameters

R is the source relation. Must implement `RelationImpl` (§4.2.3).

F represents the condition to be selected for. Must implement `QueryFilter` (§4.6.2).

Construction

`FilterRel` objects can be obtained via the `where_eq` and `where_in` methods of the `Relation` trait (§4.2.1).

Query Plan

`FilterRel` prepends *F* to the list of filter terms in the query request, and then asks *R* to fulfill the modified query.

Implemented Traits

Table 4.15 lists the traits directly implemented for `FilterRel`.

Table 4.15: Implemented Traits for FilterRel<R, F>

Trait	Bounds	Associated Types
`RelationImpl`	*R*`: RelationImpl`	`Cols =` *R*`::Cols` `FastCols =` *R*`::FastCols`
`QueryOutput<'a>`	*R*`: QueryOutput<'a>`	`QueryRow =` *R*`::QueryRow`
`Insert<`*H*`>`	*R*`: Insert<`*H*`>`	`Req: QueryRequest<Filters =` *F*`, OrderBy = HNil>`
`Delete<`F_2`>`	*R*`: Delete<sexpr!{`*F*`;`F_2`}>`	

4.7.3 ProjectedRel Type

`ProjectedRel<`*R*`,`*H*`>` is a relation adapter that represents the projection of the relation *R* onto the header *H*.

Type Parameters

R is the source of the projection operation. Must implement `RelationImpl` (§4.2.3).

H is the `Header` that will be projected onto. Must implement `ProjectFrom<`*R*`::Cols>` (§4.4.1,4.4.3).

Construction

`ProjectedRel` instances are created via the `project` method of the `Relation` trait (§4.2.1).

Query Plan

`ProjectedRel` removes all filter and ordering terms that refer to removed columns, and then asks R to fulfill the modified query. The returned records from this subquery are then encapsulated in `Projection` (§4.5.3) instances to prevent access to the removed columns.

Implemented Traits

Table 4.16: Implemented traits for ProjectedRel<R,H>

Trait	Associated Types
`RelationImpl`	`Cols =` H`,` `FastCols =` $H \cup R$`::FastCols`
`QueryOutput<'a>`	`QueryRow = Projection<`R`::QueryRow,` H`>`

4.7.4 OrderedRel Type

`OrderedRel<`R`,`O`>` is a relation adapter that presents the records in R according to the order O.

Type Parameters

R is the source relation. Must implement `RelationImpl` (§4.2.3).

O is the default presentation order. Must implement `List`, and every element of O must implement `SortKey` (§3.1.3, 4.6.3).

Construction

`OrderedRel` instances can be created via the `order_by` method on the `Relation` trait (§4.2.1).

Query Plan

Given a query request Q, if Q`::OrderBy` is a prefix of O then a new query request Q' is generated by setting `OrderBy` to O. Otherwise, Q' is identical to Q.

The modified query Q' is then fulfilled by R.

Implemented Traits

Table 4.17: Implemented traits for OrderedRel<R,O>

Trait	Bounds	Associated Types
`RelationImpl`	`R: RelationImpl`	`Cols = R::Cols`, `FastCols = R::FastCols`
`QueryOutput<'a>`	`R: QueryOutput<'a>`	`QueryRow = R::QueryRow`

4.7.5 SubordinateJoin Type

`SubordinateJoin<R,C>` is a relation adapter that represents the extension of relation R with the relation stored in column C.

Type Parameters

R is the source relation. Must implement `RelationImpl` (§4.2.3), and C must be a member of `R::Cols`.

C is the column type which contains the subrelation. It must implement `Col` (§4.3.3), and `C::Inner` must implement `RelationImpl`. `C::Inner::Cols` must be disjoint to `R::Cols`.

Construction

`SubordinateJoin` instances can be created via the subjoin method on the `Relation` trait (§4.2.1).

Query Plan

Given a query request Q with selection filter F and presentation order O,

1. Ask R to fulfill the query Q. For each record A returned,
    1. Retrieve the relation R_2 stored in column C
    2. Ask R_2 to retrieve all records which match F. For each record B returned, the tuple `(A,B)` is a candidate result.
2. Discard candidate results which do not pass the filter F.
3. If any column in R_2 appears in O, collect and re-sort the results

Implemented Traits

Table 4.18: Implemented traits for SubordinateJoin<R,C>

Trait	Associated Types
`RelationImpl`	`Cols = R::Cols ∪ C::Inner::Cols` `FastCols = R::FastCols`
`QueryOutput<'a>`	`QueryRow = (R::QueryRow, C::Inner::QueryRow)`

4.7.6 PeerJoin Type

`PeerJoin<L,R>` is a relation adapter that represents the natural join of relations L and R.

Requirements

L and R are the two source relations of the join operation. They both must implement `RelationImpl` (§4.2.3).

The intersection of `L::Cols` and `R::Cols` must be a single column K, which must define an equality operator via the `Eq` trait.

Query Plan

Given a query request Q with selection filter F and presentation order O,

1. Determine which source relation is primary for this query:
    1. If K is a fast column of L or R, but not both, that source is secondary.
    2. If F mentions any fast columns from L or R, but not both, that source is primary.
    3. If F mentions columns from only one of L or R, that source is primary.
    4. Otherwise, L is primary.
2. Ask the primary source relation to fulfill the query Q. For each record A returned,
    1. Retrieve all records from the secondary relation which match F and contain the join key value from A. For each record B returned, `PeerJoinRow<A,B>` is a candidate result.
3. Discard candidate results which do not pass the filter F.
4. If any column not in the primary relation appears in O, collect and re-sort the results.

Implemented Traits

Table 4.19: Implemented traits for PeerJoin<L,R>

Trait	Associated Types
`RelationImpl`	`Cols = L::Cols ∪ R::Cols`
`QueryOutput<'a>`	`QueryRow = PeerJoinRow<L::QueryRow, R::QueryRow>`

4.7.7 PeerJoinRow Type

`PeerJoinRow<L,R>` is the record type returned when querying a `PeerJoin` adapter. Implements `Record` and `ExternalRecord` traits (§4.5.1-2).

Type Parameters

L and R are the record types returned by the join's source relations. They must implement the `Record` and `ExternalRecord` traits.

Implemented Traits

Table 4.20: Implemented traits for PeerJoinRow<L,R>

Trait	Bounds	Associated Types
`Record`	L: `Record` R: `Record`	`Cols = L::Cols ∪ R::Cols`
`ExternalRecord<'a>`	L: `ExternalRecord<'a>` R: `ExternalRecord<'a>`	`Cols = L::Cols ∪ R::Cols`
`Clone`	L: `Clone` R: `Clone`	

4.8 Indices

The primary tool for improving query performance in a database is the *index*. These data structures store redundant copies of information to facilitate fast retrieval, at the expense of storage space and modification performance. Memquery provides both a primary index, `BTreeIndex`, which directly stores records and a secondary index, `RedundantIndex`, which holds the redundant information alongside the source relation.

Tuning program performance by adding and removing indices is a primary task of architects, who should be familiar with the types described in this chapter. Neither

library authors nor application programmers need to know much about these types: They will usually interact with them only as generic Relation types.

4.8.1 BTreeIndex

`BTreeIndex<K,R>` stores a distinct instance of the relation *R* for each unique value of the column *K*.

Type Parameters

K is the column to index on. Must implement the `Col` and `Ord` traits (§4.3.3). Must be a member of `<R as RelationImpl>::Cols`.

R is the subrelation typerelation type responsible for storing records. Must implement the `Default`, `Insert`, and `RelationImpl` traits (§4.9.4, 4.2.3).

Construction

Empty `BTreeIndex` instances are created by calling `Default::default()`. They can then be populated via the `Insert` trait.

Query Plan

Given a query request *Q* with selection filter *F* and presentation order *O*,

1. Identify the contained *K* values which fall within the range allowed by *F* (cf. `QueryFilter::bounds_of`, §4.6.2)

2. Iterate over the corresponding subrelations (of type *R*), querying for all records which match *F*.

3. If *O* is anything other than `sexpr!{}` or `sexpr!{Asc<K>}`, collect and re-sort the results.

Implemented Traits

Table 4.21: Implemented traits for BTreeIndex<K,R>

Trait	Bounds	Associated Types
`RelationImpl`	R`: RelationImpl` $K \in R$`::Cols` K`: Ord`	`Cols =` R`::Cols` `FastCols = sexpr!{`K`}`
`QueryOutput<'a>`	R`: QueryOutput<'a>`	`QueryRow =` R`::QueryRow`
`Insert<`H`>`	R`: Insert<`H`>` $K \in H$	
`Delete<`F`>`	R`: Delete<`F`>`	
`Default`	*none*	
`Clone`	R`: Clone`	

4.8.2 RedundantIndex

`RedundantIndex<`K_2`,`K_1`,`R`>` stores a $K_2 \rightarrow K_1$ index beside a single instance of the relation type R.

Type Parameters

K_2 is the index's key column. Must implement `Col` and `Ord` traits (§4.3.3).

K_1 is the column that serves as a key to the records stored in R. Must implement `Col` and `Ord`. `RedundantIndex` is most efficient when K_1 is an indexed primary key of R, but this is not required.

R is the relation type being indexed. R must implement `RelationImpl`, and R`::Cols` must contain columns K_1 and K_2.

Construction

An empty `RedundantIndex` can be created via the `Default::default()` method. or an index for a pre-existing relation can be built via the `new()` associated function.

Associated Functions

`new(`R`)->Self`

> Constructs a new `RedundantIndex` for the given relation R. When called, iterates over all records in R to populate the index.

Query Plan

Given a query request Q with selection filter F and presentation order O,

- If F refers to a member or R`::FastCols` or does not refer to column K_2, pass the unmodified query Q to R.
- Otherwise:
  1. Identify the contained K_2 values that fall within the range allowed by F (cf. `QueryFilter::bounds_of`, §4.6.2)
  2. Collect and sort the associated K_1 values, removing duplicates
  3. For each identified K_1 value, query R for all records with that K_1 value which also match the filter F.
  4. If O is anything other than `sexpr!{}` or `sexpr!{Asc<`K_1`>}`, collect and re-sort the results.

Implemented Traits

Table 4.22: Implemented traits for RedundantIndex<K_2,K_1,R>

Trait	Bounds	Associated Types

`RelationImpl`	R`: RelationImpl`
	K_1 `∈` R`::Cols`
	K_2 `∈` R`::Cols`
	K_1`: Col + Ord`
	K_2`: Col + Ord`
		`Cols =` R`::Cols`
		`FastCols =`
		`sexpr!{`K_2`;` R`::FastCols}`

`QueryOutput<'a>`	R`: QueryOutput<'a>`	`QueryRow =` R`::QueryRow`
`Insert<`H`>`	R`: Insert<`H`>` K_1 `∈` H K_2 `∈` H	
`Default`		
`Clone`	R`: Clone` K_1`: Clone` K_2`: Clone`	

4.9 Transactions and Mutation

Because of its modular nature, any given Memquery operation may invoke code from many different modules to complete its work. For infallible read-only operations, such

as queries, this poses little practical difficulty. Mutations, on the other hand, need to leave the system in a consistent state regardless of whether they succeed or fail. Memquery thus needs some mechanism to ensure that all of component routines agree on whether or not an operation can succeed.

This is a simple case of the atomic transaction problem faced by distributed databases. Because it runs within a single process, Memquery can safely assume that neither the individual modules nor the communication links between them ever fail: The root cause of such a failure would almost certainly affect the rest of the process as well, corrupting any potential recovery procedure.

Even though the modules will never fail, any of them may reject a proposed operation as invalid. Memquery's system to handle this is based on the Presumed Commit protocol described by Mohan and Lindsay [15]. A variant of the two-phase commit protocol, all proposed changes are made immediately upon request. At the same time, an undo log is collected that can return the datastructure to its original state if a rollback proves necessary.

Operations are defined as objects that implement the `RevertableOp` trait, which includes the information necessary to undo the operation in the case that the transaction needs to be aborted. Transaction instances apply these operations to a target object, and are responsible for rolling back changes in the event of an error. The Insert and Delete traits provide an abstraction to construct revertable operations that will respectively add or remove records from a relation.

4.9.1 RevertableOp Trait

`RevertableOp<`*T*`>` indicates that `Self` is a revertable operation which acts on an object of type *T*.

Library authors need to be familiar with the details of this trait in order to implement the `Insert` and `Delete` traits for their custom relation types. Application programmers will generally work with `Transaction`s instead of interacting with `RevertableOp` directly. Architects will generally not need to interact with the transaction system.

Type Parameters

T is the type of the object to be modified.

Requirements

Given a target object *x*, there must be no observable change to *x* after either of these sequences occur:

- `self.apply(&mut` *x*`)` returns `Err(…)`.
- `self.apply(&mut` *x*`)` returns `Ok(`*y*`)`, and then *y*`.revert(&mut` *x*`)` is called.

Associated Types

`Err` is the type that will be returned if this operation is unsuccessful. It must implement the `std::error::Error` trait.

`Log` is the type that is be responsible for reverting this operation. Must implement `UndoLog<T>`

Methods

`fn apply(Self, &mut T)->Result<Self::Log, Self::Err>`

> Attempts to modify the given *T*. If successful, returns `Ok(Self::Log)`. If unsuccessful, *T* is unchanged and `Err(Self::Err)` is returned.

Example Implementation

```
/// Removes the last element from a vector
struct Pop;
struct RevertPop<T>(T);
struct EmptyVec;

impl Error for EmptyVec { ... }

impl<T> RevertableOp<Vec<T>> for Pop {
    type Err = EmptyVec;
    type UndoLog = RevertPop<T>;

    fn apply(self, vec:&mut Vec<T>)
       ->Result<RevertPop<T>, EmptyVec>
    {
        match vec.pop() {
            None => Err(EmptyVec),
            Some(x) => RevertPop(x)
        }
    }
}

impl<T> UndoLog<Vec<T>> for RevertPop<T> {
    fn revert(self, vec: &mut Vec<T>) {
        vec.push(self.0);
    }
}
```

4.9.2 UndoLog Trait

`UndoLog<T>` indicates that `Self` is capable of reversing the change caused by a prior `RevertableOp<T>`.

Library authors need to be familiar with the details of this trait in order to implement the `Insert` and `Delete` traits for their custom relation types. Application programmers

will generally work with Transactions instead of interacting with the UndoLog trait directly. Architects will generally not need to interact with the transaction system.

Type Parameters

T is the type of the object that was modified.

Requirements

See the requirements for `RevertableOp`, §4.9.1

Methods

`revert(self, &mut T)`

> The given *T* must be in an equivalent state as it was immediately after the corresponding `RevertableOp` was applied. Restores the *T* to a state equivalent to immediately prior to the `RevertableOp` was applied.

Implementations

The transaction system defines `UndoLog<T>` implementations for several types, which serve as combinators. These are listed in Table 4.23.

Table 4.23: UndoLog<T> combinators

Type	Revert Behavior
`()`	Nothing to revert
`(A, B)`	Reverts *A* and then reverts *B*
`Option(A)`	If `Some`, reverts *A*
`Vec<A>`	Reverts all elements in reverse order

Both A and B must implement UndoLog<T>

4.9.3 Transaction Type

`Transaction<'a, Target, Log, Err>` represents a multi-step atomic change which is currently in progress. Note that most methods on `Transaction` consume `Self` and produce a new `Transaction` instance with different type parameters.

Both application programmers and library authors will need to be familiar with the `Transaction` type, but architects will generally not interact with the transaction system.

Invariants

A transaction is always in one of two states, *active* or *poisoned*.

When *active*, all of the attempted operations have succeeded, and a commit request will succeed.

When *poisoned*, some previously attempted operation failed. The transaction target has been restored to a state equivalent to when the transaction was started. All future attempted operations on this transaction will be ignored.

Type Parameters

`'a` is the region in which the transaction has exclusive access to *Target*.

Target is the type that is being modified by this transaction.

Log is an `UndoLog` which will return *Target* to the state it was in at the beginning of the transaction.

Err is the error type that will be returned from `commit()` if this transaction becomes poisoned.

Associated Functions

`start(&'a mut` *Target*`)->Transaction<'a,` *Target*`, (), Infallible>`

> Starts a new transaction.

Methods

`commit(Self)->Result<(), Err>`

> Confirms the transaction's changes. If the transaction is active, returns `Ok(())`. If poisoned, returns the error which caused the poisoning.

`revert(Self)`

> Discards the transaction's changes. If the transaction is active, returns the target to its initial state. If poisoned, this is a no-op.

`inspect(&Self)->Option<&`*Target*`>`

Allows the inspection of the changes which will be committed. Returns `None` if the transaction is poisoned.

This can be used, for example, to perform an atomic commit of multiple concurrent transactions:

```
let mut relation_a: A = …;
let mut relation_b: B = …;

let tx_a = Transaction::start(&mut relation_a).apply(…);
let tx_b = Transaction::start(&mut relation_b).apply(…);

fn check_constraints(&A, &B)->bool { … }

match (tx_a.inspect(), tx_b.inspect()) {
    (Some(a_ref), Some(b_ref))
    if check_constraints(a_ref, b_ref) => {
        // These commits will never fail
        tx_a.commit().unwrap();
        tx_b.commit().unwrap();
    }
    (_,_) => {
        // Either the constraint check failed or
        // one of the transactions is poisoned.
        tx_a.revert();
        tx_b.revert();
    }
}
```

`apply<`*Op*`>(Self, `*Op*`)->Transaction<'a, `*Target*`, …>`

Op must implement `RevertableOp<`*Target*`>`. If the transaction is active, applies the given operation to *Target*.

If *Op* fails or `Self` is poisoned, returns a poisoned transaction.

`apply_multi<`*Op*`>(Self, impl IntoIterator<Item=`*Op*`>)`
`  ->Transaction<'a, `*Target*`, …>`

Op must implement `RevertableOp<`*Target*`>`. If the transaction is active, applies each operation yielded by the iterator to Target. If any operation fails, the transaction becomes poisoned and no further operations are attempted.

`subtransaction<`*Access*`, `*Body*`>(Self, `*Access*`, `*Body*`)`
`  ->Transaction<'a, `*Target*`, …>`

If the transaction is active, executes a transaction on a component of *Target*.

Access must be a closure with the signature `fn(&mut` *Target*`)->&mut` *Inner*.

Body must be a closure with the signature
`fn<'a>(Transaction<'a, `*Inner*`, …>) -> Transaction<'a, `*Inner*`, …>`.

Access is called immediately to obtain the root value for the subtransaction, which is then passed as an argument to *Body*. If the transaction returned from *Body* is active, both *Access* and its undo log are stored to facilitate a future revert operation.

If the transaction returned from Body is potentially reverted at a future time. If poisoned, then `Self` is reverted and becomes poisoned.

```
struct Schema {
    a: RelA,
    b: RelB,
    …
}

let mut db: Schema = …;

Transaction::start(&mut db)
    .subtransaction(
        |schema| &mut schema.a,
        |tx| tx.apply(RelA::insert_op(…)))
    .subtransaction(
        |schema| &mut schema.b,
        |tx| tx.apply(RelB::insert_op(…)))
    .commit().unwrap()
```

`into_undo_log(Self)->Result<`*Log*`, `*Err*`>`

Destroys the transaction without either committing or reverting it. If the transaction is active, returns `Ok(`*Log*`)`. If poisoned, returns `Err(`*Err*`)`.

This method is primarily used by library authors to implement a RevertableOp that performs a multi-step operation:

```
impl<T> RevertableOp<T> for MyOp {
    type Err = …;
    type Log = …;

    fn apply(self, x:&mut T)->Result<Self::Err, Self::Log> {
        Transaction::start(x)
             .apply(…)
             .apply_multi(…)
             .into_undo_log()
    }
}
```

4.9.4 Insert Trait

`Insert<`*H*`>` indicates that `Self` is capable of adding records with header *H*.

Application programmers need to be familiar with how to use the `Insert` trait and library authors should be familiar with implementing it. Architects need to know which relation types implement `Insert`, but do not need to know details of how it is used.

Type Parameters

H is the relational header of records to be inserted. Must implement `Header` (§4.4.1).

Associated Types

`Remainder` contains the columns of *H* that are not consumed by the insert operation. Must implement `Header`.

`Op` is the operation type that will actually perform the insert. Must implement `RevertableOp<Self>`.

Associated Functions

`insert_op<`*R*`>(`*R*`)->(Self::Op, Self::Remainder)`

> Constructs an operation that will insert the record *R* into `Self`. This operation can then be applied to `Self` via a transaction.
>
> *R* must implement `Record<Cols = `*H*`>` (§4.5.1).

```
col!{A:usize}
col!{B:usize}
col!{C:usize}

type Rel = Vec<(A,C)>;

let mut rel: Rel = vec![];

let (op, remainder) = Rel::insert_op((C(42), B(3), A(7));
Transaction::start(&mut rel).apply(op).commit().unwrap();

(rel, remainder.project::<B>())
⟹ (vec![(A(7), C(42))], B(3))
```

Methods

`insert<`*R*`>(&mut Self, `*R*`)->Result<(), Self::Err>`

> Inserts the record *R* into `Self`. *R* must implement `Record<Cols = `*H*`>` (§4.5.1).
>
> A default implementation is provided, which is roughly equivalent to the following code (error types differ):

```
let (op, _) = Self::insert_op(record);
Transaction::start(self).apply(op).commit()
```

`insert_multi<`*I*`>(&mut Self, `*I*`)->Result<(), Self::Err>`

> *I* must implement `IntoIterator`, and the items yielded by *I* must be records that implement `Record<Cols=`*H*`>` (§4.5.1).
>
> Inserts the yielded records into Self. Returns `Ok(())` if the operation is successful. If any individual insert fails, all prior inserts are reverted and `Err(...)` is returned; no further records are retrieved from the iterator.
>
> A default implementation is provided, which is roughly equivalent to the following code (error types differ):

```
let ops = records.into_iter().map(|r| Self::insert_op(r).0);
Transaction::start(self).apply_multi(ops).commit()
```

4.9.5 Delete Trait

`Delete<F>` indicates that `Self` is capable of removing records. It is usually invoked via the `Relation::truncate` method (§4.2.1).

Application programmers should be familiar with how to use the `Delete` trait and library authors should be familiar with implementing it. Architects need to know which relation types implement `Delete`, but do not need to know details of how it is used.

Type Parameters

F specifies the records to be removed. It must implement the QueryFilter trait (§4.6.2).

Associated Types

`Op` is the operation type that will actually perform the removal. Must implement `RevertableOp<Self>` (§4.9.1).

Associated Functions

`delete_op(`*F*`)->Self::Op`

> Constructs an operation that will remove all records that match the filter *F* from `Self`. This operation can then be applied to `Self` via a transaction:

```
col!{A:usize}
col!{B:usize}

type Rel = Vec<(A,B)>;

let mut rel = vec![
    (A(1), B(2)),
    (A(2), B(5)),
    (A(3), B(2)),
];

Transaction::start(&mut rel)
    .apply(Rel::delete_op(Exact(B(2))))
    .commit()
    .unwrap();

rel
⟹ vec![(A(2), B(5))]
```

5 Case Study

The fundamental claim that both Codd and Parnas make is that it is unwise to intertwine code that specifies what information is needed with code that specifies how to retrieve it [2,1]. They each present an example problem and several different data storage models that could be used to implement a solution. Codd's problem is a simple manufacturing inventory management system, and Parnas' problem is writing a text indexing program. They then claim, plausibly but without direct evidence, that programs written directly against these various data models must necessarily be quite different to each other. This hinders the ability of maintenance programmers to adjust the underlying storage model for changing operational conditions.

This study translates Codd's example data models and sample query into Rust programs. It consists of two batteries of implementations: One based on Rust's standard library collections, comparable to most modern languages', and another based on Memquery. Each battery contains implementations that correspond to the various storage models originally proposed by Codd.

A common goal of software re-engineering efforts is to improve the program's performance. To this end, each implementation's query performance is measured. A significant difference in performance indicates that there is a practical reason to prefer one structure over another.

It is important to note, however, that the best performance does not necessarily indicate the best implementation. Factors not measured here, such as code maintainability, may make a less-performant implementation more desirable in practical use. Also, the study measure the performance of only one operation. In practice, a program's data model needs to accommodate a wide variety of different operations. Altering the model to improve the performance of one operation often hinders the performance of others. The performance each operation has a different effect on the overall program utility: Improving the performance of a frequent operation has more practical impact than improving the performance of a rare one.

5.1 Methodology

To illustrate the problems caused by hierarchical data modeling, Codd provides an example of an inventory management system. The system is required to track a number of projects and their component parts, where some parts may be needed for several different projects. He then describes five different hierarchical models that satisfy the original requirements.

He asserts that each of these models requires a different algorithm to perform a sample query: Listing the part number, part name, and quantity committed to a particular project. Instead of directly supporting this assertion, he invites readers to convince themselves by writing sample programs for each of the provided models. This study does just that.

5.1.1 Implementation Strategy

Each of Codd's models describe the fields that should be present in each table and how the tables are related to each other. All of Codd's data models contain a numeric id for each part and project, along with several additional fields. There is one field that is a property of the part-project pair: the quantity of the part that is commited to the specified project.

Because Codd doesn't describe what indexing strategy should be used, this study considers two submodels for each: One that indexes solely by the integer ids and another that indexes specifically on the fields used in the tested query. Error: Reference source not found summarizes the tested models.

For the control implementations, each tables is stored as a `BTreeMap` that contains a custom struct. This struct's fields directly correspond to the fields that Codd specifies. Where Codd specifies that one table is "subordinate to" another, it is stored inside the latter table's structure. Otherwise, the top-level tables are stored as fields in an `Inventory` structure, which represents the entire data store.

Structures 1-4 have directly comparable Memquery implementations to their counterpart control implementations. The Memquery implementations of Structure 5, however, differ slightly in order to better represent real-world usage: In both 5a and 5b, the commitment relation is defined with a primary index on `PartId` and a secondary index on `ProjectId`. For 5b, the projects relation also has a secondary index on `PartName`. All of the schema definitions are listed in Appendix B.

The query function is a method on the Inventory structure. It takes a single argument, the project name to report on, and returns an iterator of `QueryResultRef` values. These values contain the three data items that Codd requested to be printed out: the part id, part name, and quantity of parts committed to the requested project.

Table 5.1: Summary of control implementations

Description	Structure	Part Index	Project Index	Commitment Index
Projects subordinate to parts	1a	Id	Id	
	1b	Id	Name	
Parts subordinate to projects	2a	Id	Id	
	2b	Id	Name	
Parts and projects as peers, commitment relationship subordinate to projects	3a	Id	Id	Part Id
	3b	Id	Name	Part Id
Parts and projects as peers, commitment relationship subordinate to parts	4a	Id	Id	Project Id
	4b	Id	Name	Project Name
Parts, projects, and commitment relationship as Peers	5a	Id	Id	(Part Id, Project Id)
	5b	Id	Id	(Project Name, Part Id)

Indices in some Memquery implementations vary; see the main text for details

5.1.2 Test Data and Evaluation

The datasets used for this study are synthetic. The necessary randomness is supplied by a 12-round ChaCha stream cipher seeded with the first 256 bits of π [16,17]. The test data consists of randomly-generated project, part, and commitment records. On average, each part is used by two different projects. To minimize potential insert-ordering effects, each of these three record lists is shuffled prior to being presented to the implementations' loading routines.

A sequence of 100 project names to query is also generated. These are all present in the test data. For each implementation, the function under test consists of running each of these 100 queries to completion and discarding the results. The query method signature has been designed to not require any heap allocations to be performed during the measurement phase of the performance tests.

5.1.3 Performance Measurement

All of the implementations are present in the same executable and measured within a single process invocation. This executable is compiled with Rust's default release-mode settings, which includes compiler optimizations. The various datastores are all initialized before any test runs are made and remain in memory throughout the testing procedure.

The measurements themselves are collected by the Criterion benchmarking system [18]. Criterion tests are not interleaved: Each function is tested to completion before any other functions are tested. This introduces a potential for bias to be introduced: If the ambient system load changes while the test program is running, the later tests will not be comparable to the earlier ones.

Several steps have been taken to mitigate this risk: The number of programs running at the same time as the benchmarking program is minimized. While the benchmarking program is running, the operator monitors system activity using the top program. The benchmarking program is configured to run the entire test suite twice in succession. If an anomaly is noted by either of these last two measures, the entire run is rejected as invalid.

Each function's test proceeds in 3 phases: warmup, measurement, and analysis. The warmup phase runs the test function constantly for 60 seconds. This allows the hardware's predictive systems to adapt themselves to the workload that will be measured. Criterion also uses this opportunity to make a rough estimate of the final result.

During the measurement phase, 50 timed samples are collected. Each sample involves running of several iterations of the test function. This helps mitigate problems that arise when a single iteration is fast compared to the available timer's resolution. The particular number of iterations is chosen based on the preliminary estimate collected during the warmup phase, with a target duration of 12 seconds per sample. The analysis phase then calculates and prints statistics about the function's runtime.

5.2 Performance Results

The measurements were collected on a computer with a 2.8 GHz Intel Core i7-7700HQ CPU and 64-bit wide DDR4-2400 memory. Figure 5.1 shows the mean elapsed time required for each of the 20 study implementations on the three different synthetic datasets. 5.1(a) shows the Memquery-based implementations and 5.1(b) shows the control implementations. With the exception of structure 5a, which will be explored in the discussion section, each Memquery implementation has a similar asymptotic behavior as its counterpart control implementation. As described above, the entire test battery was run twice. The 90^{th} percentile relative error for any data point between the two runs was 1.9%, with a maximum of 5.3%.

Figure 5.2 shows the minimum, mean, and maximum observed ratios between Memquery-based query times and the corresponding control times. Structure 5a has been excluded from these statistics and displayed separately. As the database size grows, there is a noticeable reduction in the runtime overhead imposed by Memquery.

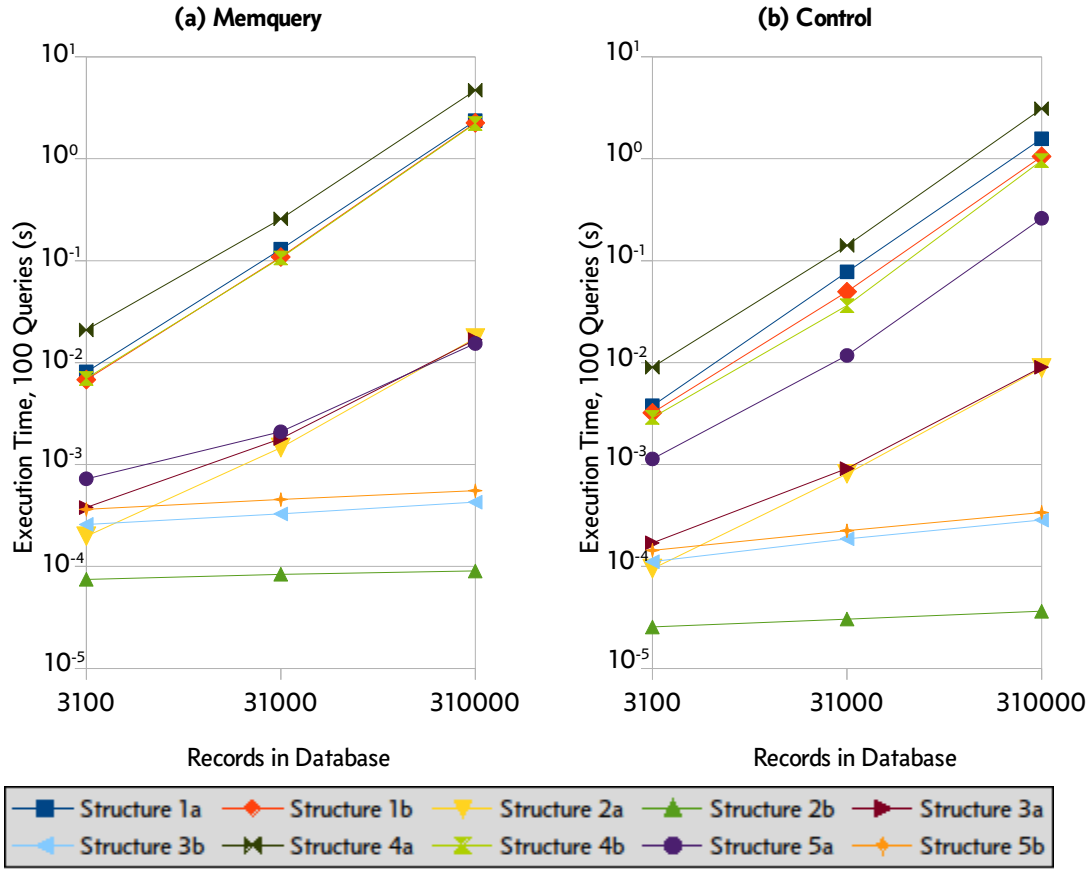

Figure 5.1: Execution time of Memquery and control implementations

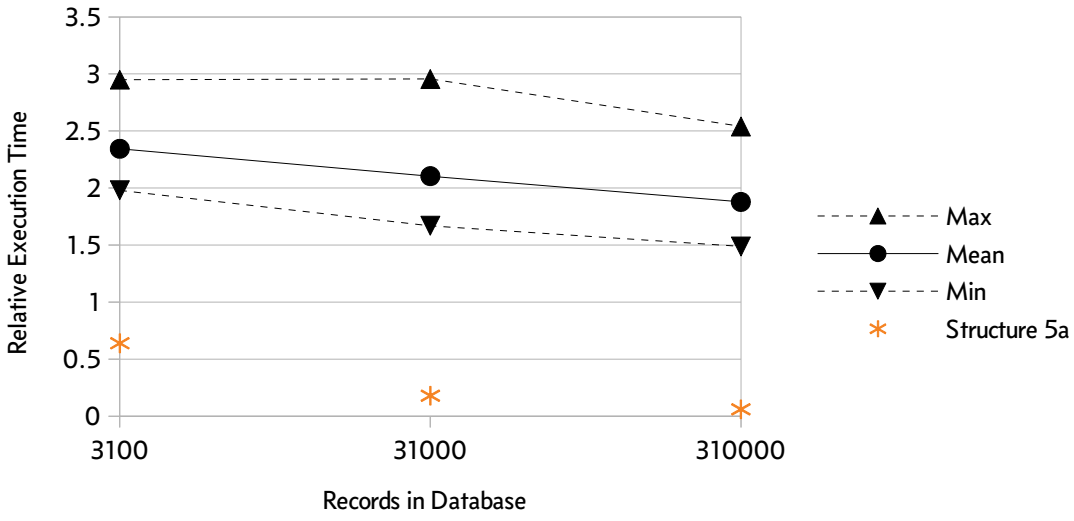

Figure 5.2: Runtime overhead of Memquery vs. control implementations

5.3 Discussion

Memquery shows a 2-3x performance penalty compared to the control implementations in almost all cases. The discrepancy in the performance of structure 5a is a result of differing query plans between the control and Memquery implementations. Because the commitment table is indexed first by `PartId` in the control implementation, it must perform a full scan of the table to find the entries for the requested project. The Memquery implementation, on the other hand, correctly uses its secondary index to locate the relevant records without inspecting the entire table. Maintaining this kind of redundant information in hand-written code is possible, but error-prone.

Selecting the wrong model, however, can incur a penalty of several orders of magnitude. If Memquery's design allows programmers to use a more appropriate model than they otherwise would, the corresponding performance increase easily dominates the runtime overhead imposed by Memquery. The ability to choose an appropriate model depends on both experience and the programmer's ability to experiment with different options.

Listing 5.1 shows the query method body for each of the Structure 2 implementations. The only difference between Structures 2a and 2b is the projects table index. This change results in a 100x difference in execution speed on the largest dataset. For Memquery, updating the schema definition is enough: the query method requires no changes. This is a feature common to all of the Memquery implementations: Changing the indices defined for a relation will automatically change the performance characteristics of any queries made against that relation.

The control implementations, on the other hand, have some substantive differences. For 2a, it iterates over the entire projects table, filters those results by the provided project name, and then queries the parts table for each matching entry. 2b, on the other hand, does a direct lookup of the project name, which returns an `Option` instead of an iterator. It then uses `Option::map` to conditionally query the parts table if a project was found. It is interesting to note that the 2a code would continue to work properly in this case, but would not take advantage of the changed index. If the programmer neglects to make this change, the only way to detect the error would be to notice that the expected performance increase didn't occur.

Listing 5.1: Query implementations for Structure 2

```
// Structure 2a, control
self.projects
    .values()
    .filter(move |proj| proj.name == project)
    .for_each(move |proj| {
        proj.parts.values().for_each(|part| out(QueryResultRef {
            part_id: part.id,
            part_name: &part.name,
            quantity_committed: part.quantity_committed,
        }));
    });

// Structure 2b, control
self.projects
    .get(project)
    .map(move |proj| {
        proj.parts.values().for_each(|part| out(QueryResultRef {
            part_id: part.id,
            part_name: &part.name,
            quantity_committed: part.quantity_committed,
        }));
    });

// Structure 2a and 2b, Memquery
self.projects.by_ref()
    .subjoin::<ProjectParts>()
    .where_eq(&ProjectName(String::from(project)))
    .iter_as::<QueryResultRef>()
    .for_each(out)
```

Though the most performant model for this query, structure 2b provides no access to part information except by iterating through the projects table. If this becomes a problem for other queries, it may be necessary to change this relationship. Structure 3b moves the parts table to the top level of the schema and leaves only the commitment information inside the projects table. These query implementations are shown in Listing 5.2.

Here, the Memquery implementation needs to be adjusted to reflect the new table layout. Instead of a single subordinate join on the `ProjectParts` field, it performs a subordinate join on the `ProjCommits` field followed by a natural join on the `Parts` table. The remainder of the query code is identical to that of structures 2a and 2b. If this change is forgotten, the compiler will reject the `iter_as` call due to missing columns. It is also possible to define a method responsible for performing these joins which returns the resulting view, which can be shared among all queries that need this information.

The control implementation needs to add an additional level of iteration with this change, which increases the method length by approximately 50%. Unlike the

Memquery change, there is also no obvious way to extract a method that would reduce the burden of making corresponding changes to additional query routines.

Listing 5.2 Query implementations for Structure 3b

```
// Structure 3b, control
self.projects
    .get(project)
    .into_iter()
    .for_each(|proj| {
        proj.parts_committed
            .iter()
            .for_each(|
                (&part_id, &quantity_committed)
            | out(QueryResultRef {
                        part_id,
                        part_name: &self.parts[&part_id].name,
                        quantity_committed,
                    }))
            })

// Structure 3b, Memquery
self.projects.by_ref()
    .subjoin::<ProjCommits>()
    .join(self.parts.by_ref())
    .where_eq(&ProjectName(String::from(project)))
    .iter_as::<QueryResultRef>()
    .for_each(out)
```

Changing code between any two of these data models shows similar characteristics: the hand-written code is all in a similar style, but requires changes to be made throughout the query method. There is no single recipe for determining these changes; each one must be individually engineered based on the details of the storage layout. The Memquery implementations, on the other hand, all have the same structure. They build a join view that contains all of the needed columns and then filter and project that view to describe the query. Though the specification of the join view varies between data models, the filtering and projection code is identical between all 10 implementations.

6 Limitations and Future Work

As presented here, Memquery sets down a framework that can be expanded into a generally-useful library. It has a number of deficiencies, however, that make it unsuited for production use in its current form.

Some of these deficiencies remain unknown but can be discovered through further research. Most modern software architecture literature implicitly assumes that a program's data model will follow object-oriented design. There are significant differences between relational and object-oriented data modeling. Like every other software engineering technique, relational modeling isn't a silver bullet. Investigating how to integrate relational modeling into current software development practices will inevitably suggest a number of improvements to Memquery's design.

Other deficiencies have presented themselves during the course of Memquery's development. The most prominent of these are discussed here, along with potential strategies to correct them.

6.1 Cross-Relation Constraints

It is often useful to maintain invariants that involve related records of different types. At present, Memquery provides only limited support for enforcing these. Changes to individual relations can be held in a transaction until the invariant is checked, but there is no inherent mechanism to force this check to happen in every case.

In order to provide this guarantee, Memquery's transaction system needs to be extended to include a verification function. This function would be run during the commit request, and would be given the opportunity to abort the transaction. There should also need be some record of the changes made during the transaction. This would allow the verification function to inspect only those records that were actually changed.

As part of this work, it probably will be useful to formalize the concept of a schema which contains multiple relations. The schema would then only provide access to transactions with a verification function that enforces the schema's constraints.

6.2 Adaptation to Other Programming Languages

Languages like C++, C#, and Java are still the dominant choice for systems and application programming. Additionally, several popular deployment targets are closely

integrated with specific programming languages, such as Swift for iPhone applications and ECMAScript for web applications. While Rust's features make it a good choice for prototyping Memquery, its audience will remain limited unless it is usable from these other languages.

The major challenge here is Memquery's reliance on static dispatch in Rust's expressive type system. Porting Memquery to another language would likely require either a compiler extension or heavier reliance on runtime dispatch, which incurs a performance penalty. Alternatively, the Rust version of Memquery could be compiled into a system library that can be linked into applications written in other programming languages. The wide variety of types used internally by Memquery would, however, make it difficult to specify an appropriate interface. A final option would be to write a code generation tool capable of searching a codebase for queries, which then emits a suitable implementation for each query that is found.

6.3 Execution Speed

As shown in the case study (§5), Memquery is 2-3 times slower than the equivalent hand-written code. The maintenance benefits provided by Memquery may justify this in many cases, but there is no inherent reason for the overhead to be this high. Careful profiling should be able to determine where this slowdown occurs. It can then be corrected by either altering Memquery's code or by improving the Rust compiler.

Another potential avenue for improving performance is parallelization. All queries are currently performed on a single processor core. Because queries are a read-only operation, it should be possible to dispatch some of the work to additional threads, and then collect the results.

6.4 Additional Relation Types

All of the relations provided by Memquery are based on Rust's `Option`, `Vec`, and `BTreeMap` types. While these are sufficient to demonstrate the efficacy of Memquery's approach, there are many more options that could be considered.

Copy-on-write shared-structure collections, like those provided by the im Rust crate, could be particularly interesting to explore. They could provide an inexpensive snapshot functionality, which could in turn be used to implement non-blocking transactions. Some threads, like those driving a user interface, could work from a recent snapshot while others are busy performing updates.

Additionally, relations could be added for indexing specialty datasets. Geospatial data have unique query requirements, for example, which are not a good match for the current one-column-at-a-time model. Adding support for a geospatial index would require defining a new query filter type and a corresponding index relation that can

process it efficiently. This would enable Memquery to efficiently query relations that contain mixed data: Queries containing a geospatial filter term will be served by the specialty index, and all other queries can be served by Memquery's built-in indices and relations.

6.5 Additional Relational Operators

Memquery currently has no support for two of the relational operators that Codd described in his original paper: Column renaming and cyclical joins. Adding support for column renaming would be relatively straightforward, but involves a significant amount of work: A suitable rename operator needs to be defined for columns, records, query filter terms, and sort keys. Once these have been developed, a relation adapter to rename a column A to column B can be written which renames B to A in incoming query requests, and then A to B in the results.

Cyclical joins rely on the gamma operator, which selects records where two different columns A and B contain the same value. Defining a filter term which tests for this condition is trivial, but optimizing it is not. One approach would be to define a value-binding operation for filter terms. Then, when a query plan knows that the value of A will be constant for all results from a subquery, it can pass that information to the subquery's planner. The subquery planner can, in turn, use this information to reduce the candidate rows that it considers.

6.6 Aggregate Queries

Some common database operations, like grouping and aggregation, are not directly supported by Memquery: Queries only return references to pre-existing records. To perform these operations in the current version of Memquery, the user needs to select the involved records and then manually perform the aggregation. This hinders the use of aggregate results as relations in further calculations.

The primary blocker for building a relation adapter to represent these queries is the lack of a place to store generated values in a query result. One approach would be to cache the generated values inside the adapter itself, but properly evicting unneeded results from the cache may prove difficult.

6.7 Join Efficiency

Currently, querying a join always dispatches a subquery to both source relations. There are some cases where this is not necessarily required, however: Consider a join between relations A and B, on column K, where every record in B has a unique value for K. If all of the requested columns are present in A and A contains no K values that are absent from B, there is no need to inspect B to produce the results.

There are two potential ways to handle this situation in Memquery. The first is to include a list of output columns in the query request and allow each query to return a different record type. This is how Memquery was originally designed; for anything except the most trivial cases, it proved intractable for Rust's current trait solver. Improvements to the trait solver could make this approach viable in the future.

The other is for this foreign-key join to return a record type that defers querying B until one of its columns is requested. This should be viable without changes to the compiler, but may introduce undesirable performance variability. At the moment, retrieving a column value from a returned record is always a trivial operation. Adopting this strategy would make the first retrieval of a column from *B* take significantly longer than any other similar operation, as it would need to find and cache the corresponding record in *B*.

7 Conclusion

There are many different strategies for organizing application data in memory, and each has both benefits and drawbacks. A simple unordered array, for example, can both add new records quickly and efficiently iterate over all records. Accessing a particular record, however, is quite inefficient. If the array is sorted, looking up a record by its sort key is efficient, but adding a new record requires moving existing ones to make space. These are by no means the only two approaches available: A staple of computer science research is to develop and analyze new strategies for storing and retrieving data.

Every application has its own set of operational constraints, which means that the correct strategy for one application may be different from that of another. Often, the full set of operational constraints is not known during development. Instead, some can only be discovered by observing the program *in situ*. Further, the operational constraints placed on a program can change over time as users modify their own behavior in response to the program.

In order to adapt a program to these changing circumstances, it may be necessary to revisit the storage strategy that the program uses internally. If these various storage strategies can all present a common interface to other components of the program, replacing one with another will be both cheaper and less error-prone.

Relational algebra describes such a common interface. Over a half-century of use in databases, it has proven to effectively balance the needs of all stakeholders. Users can retrieve the data they need without understanding how it is organized on disk, Administrators can reorganize data to meet changing demands, and database system designers can invent new ways of organizing data for better performance. Each of these three activities can occur independently, without coordination with the others.

By adopting the relational algebra model, Memquery brings these advantages to general-purpose programming. Application programmers can access the data they need, architects can reorganize the program's data storage to meet changing demands, and library authors can develop new tools to improve performance.

References

[1] D. L. Parnas. 1972. “On the criteria to be used in decomposing systems into modules.” *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. DOI:https://doi.org/10.1145/361598.361623

[2] E. F. Codd. 1970. “A relational model of data for large shared data banks.” *Commun. ACM* 13, 6 (June 1970), 377–387. DOI:https://doi.org/10.1145/362384.362685

[3] Jeremy Gibbons, Fritz Henglein, Ralf Hinze, and Nicolas Wu. 2018. “Relational algebra by way of adjunctions.” *Proc. ACM Program. Lang. 2, ICFP*, Article 86 (September 2018), 28 pages. DOI:https://doi.org/10.1145/3236781

[4] Erik Meijer. 2007. “Confessions of a used programming language salesman.” In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 677–694. DOI:https://doi.org/10.1145/1297027.1297078

[5] Erik Meijer. 2011. “The world according to LINQ.” *Commun. ACM* 54, 10 (October 2011), 45–51. DOI:https://doi.org/10.1145/2001269.2001285

[6] Oren Eini. 2011. “The pain of implementing LINQ providers.” *Commun. ACM* 54, 8 (August 2011), 55–61. DOI:https://doi.org/10.1145/1978542.1978556

[7] Microsoft Corp. 2021. *.NET 5.0 documentation.* Retrieved from https://docs.microsoft.com/en-us/dotnet/

[8] Ashley Williams. 2021. *Hello, World!* Retrieved from https://foundation.rust-lang.org/posts/2021-02-08-hello-world/

[9] Steve Klabnik and Carol Nichols, et al. *The Rust Programming Language.* Retrieved from https://doc.rust-lang.org/stable/book/

[10] The Rust Project Developers. *The Rustonomicon: The Dark Arts of Unsafe Rust.* Retrieved from https://doc.rust-lang.org/nomicon/

[11] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. “RustBelt: securing the foundations of the Rust programming language.” *Proc. ACM Program. Lang. 2, POPL*, Article 66 (January 2018), 34 pages. DOI:https://doi.org/10.1145/3158154

[12] John McCarthy. 1978. “History of LISP.” *SIGPLAN Not. 13, 8* (August 1978), 217–223. DOI:https://doi.org/10.1145/960118.808387

[13] Paho Lurie-Gregg. *Typenum 1.13.0 documentation*. Retrieved from https://docs.rs/typenum/1.13.0/typenum/

[14] P. Leach, M. Mealling, and R. Salz. 2005. *RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace*. RFC Editor, USA

[15] C. Mohan and B. Lindsay. 1983. "Efficient commit protocols for the tree of processes model of distributed transactions." In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83)*. Association for Computing Machinery, New York, NY, USA, 76–88. DOI:https://doi.org/10.1145/800221.806711

[16] Daniel J. Bernstein. 2008. "ChaCha, a variant of Salsa20." *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*.

[17] "Pi in hexadecimal." *The On-Line Encyclopedia of Integer Sequences*, published electronically at https://oeis.org, 2001, Sequence A062964

[18] Brook Heisler, et al. *Criterion.rs documentation*, retrieved from https://bheisler.github.io/criterion.rs/book/index.html

[19] Moses Schönfinkel. 1924. "Über die Bausteine der mathematischen Logik." *Mathematische Annalen* 92, pp. 305–316. Translated by Stefan Bauer-Mengelberg as "On the building blocks of mathematical logic" in Jean van Heijenoort, 1967. *A Source Book in Mathematical Logic, 1879–1931*. Harvard Univ. Press: 355–66.

Appendix A: Turing Completeness of Rust's Type System

One way to demonstrate that Rust's type system is Turing-complete is by implementing the SK basis combinators [19]. Such an implementation is given below: All valid SK expressions implement the `Derive` trait, which produces the expression's normal form as `Derive::Result`. If an expression does not have a normal form or is exceptionally complicated to resolve, the compiler will halt when its internal recursion limit is reached.

Each implementation of the `DeriveStep` trait describes a single derivation step. The `DeriveStep::Continue` type indicates whether or not any derivations were made. The `DeriveCont` trait then uses this information to iterate until a fixed point is reached.

```
struct True;
struct False;

struct S;
struct K;
type I = ((S,K),K);

trait DeriveStep {
    type Continue;
    type Result;
}

trait Or { type Result; }

impl Or for (False, False) { type Result = False; }
impl Or for (True, False) { type Result = True; }
impl Or for (False, True) { type Result = True; }
impl Or for (True, True) { type Result = True; }

// No arguments
impl DeriveStep for S {
    type Continue = False;
    type Result = S;
}

impl DeriveStep for K {
    type Continue = False;
    type Result = K;
}

// One argument
impl<X> DeriveStep for (S,X)
    where X: DeriveStep
{
    type Continue = X::Continue;
    type Result = (S, X::Result);
```

```
}

impl<X> DeriveStep for (K,X)
    where X:DeriveStep
{
    type Continue = X::Continue;
    type Result = (K, X::Result);
}

// Two arguments
impl<X,Y> DeriveStep for ((K,X), Y)
    where X:DeriveStep
{
    type Continue = True;
    type Result = X::Result;
}

impl<X,Y,Cont> DeriveStep for ((S,X), Y) where
    X:DeriveStep,
    Y:DeriveStep,
    (X::Continue, Y::Continue): Or<Result=Cont>
{
    type Continue = Cont;
    type Result = ((S, X::Result), Y::Result);
}

// Three arguments
impl<X,Y,Z> DeriveStep for (((S, X), Y), Z) where
    X:DeriveStep,
    Y:DeriveStep,
    Z:DeriveStep
{
    type Continue = True;
    type Result = ((X::Result, Z::Result),
                   (Y::Result, Z::Result));
}

impl<X,Y,Z> DeriveStep for (((K,X), Y), Z) where
    X:DeriveStep,
    Z:DeriveStep,
{
    type Continue = True;
    type Result = (X::Result, Z::Result);
}

// 4+ Arguments
impl<V,W,X,Y,Z,A,ACont,ZCont,Cont> DeriveStep for ((((V,W), X), Y), Z)
where
    (((V,W), X), Y): DeriveStep<Result = A, Continue=ACont>,
    Z: DeriveStep<Continue = ZCont>,
    (ACont, ZCont): Or<Result = Cont>,
{
    type Continue = Cont;
    type Result = (A, Z::Result);
}

trait DeriveCont<Expr> { type Result; }
impl<Expr> DeriveCont<Expr> for False { type Result = Expr; }
impl<Expr> DeriveCont<Expr> for True where
```

```
    Expr: DeriveStep,
    Expr::Continue: DeriveCont<Expr::Result>
{
    type Result = <Expr::Continue as DeriveCont<Expr::Result>>::Result;
}

trait Derive { type Result; }

impl<Expr> Derive for Expr where True: DeriveCont<Expr> {
    type Result = <True as DeriveCont<Expr>>::Result;
}
```

Appendix B: Case Study Schemas

This chapter contains the Memquery schema definitions used in the case study (§5). Ellipses indicate omitted implementation details.

Common Definitions

```
col!{ pub PartId:       usize  }
col!{ pub PartName:     String }
col!{ pub PartDesc:     String }
col!{ pub QtyOnHand:    usize  }
col!{ pub QtyOnOrder:   usize  }
col!{ pub ProjectId:    usize  }
col!{ pub ProjectName:  String }
col!{ pub ProjectDesc:  String }
col!{ pub QtyCommitted: usize  }

pub struct Project { … }
pub struct Part { … }
pub struct Commit { … }
pub struct QueryResultRef<'a> { … }

impl Record for Project {
    type Cols = sexpr!{ProjectId, ProjectName, ProjectDesc};
    …
}

impl Record for Part {
    type Cols = sexpr!{
        PartId, PartName, PartDesc, QtyOnHand, QtyOnOrder
    };
    …
}

impl Record for Commit {
    type Cols = sexpr!{PartId, ProjectId, QtyCommitted};
    …
}

impl<'a> FromExternalRecord<'a> for QueryResultRef<'a> {
    type Cols = sexpr!{PartId, PartName, QtyCommitted};
    …
}
```

Structure 1a: Projects Subordinate to Parts

```
col!{ PartProjects:
  RelProxy<Rc<
    BTreeIndex<ProjectId,
      Option<(Project, QtyCommitted)>>>>
}

pub struct Inventory {
    parts: BTreeIndex<PartId,
             Option<(Part, PartProjects)>>,
}
```

Structure 1b: Projects Subordinate to Parts

```
col!{ PartProjects:
  RelProxy<Rc<
    BTreeIndex<ProjectName,
      Option<(Project, QtyCommitted)>>>>
}

pub struct Inventory {
    parts: BTreeIndex<PartId,
             Option<(Part, PartProjects)>>,
}
```

Structure 2a: Parts Subordinate to Projects

```
col!{ ProjectParts:
  RelProxy<Rc<
    BTreeIndex<PartId,
      Option<(Part, QtyCommitted)>>>>
}

pub struct Inventory {
    projects: BTreeIndex<ProjectId,
                Option<(Project, ProjectParts)>>,
}
```

Structure 2b: Parts Subordinate to Projects

```
col!{ ProjectParts:
  RelProxy<Rc<
    BTreeIndex<PartId,
      Option<(Part, QtyCommitted)>>>>
}

pub struct Inventory {
    projects: BTreeIndex<ProjectName,
                Option<(Project, ProjectParts)>>,
}
```

Structure 3a: Parts and Projects as Peers, Commits Subordinate to Projects

```
col!{ ProjCommits:
  RelProxy<Rc<
    BTreeIndex<PartId,
      Option<(PartId, QtyCommitted)>>>>
}

pub struct Inventory {
    parts:    BTreeIndex<PartId,
                Option<Part>>,
    projects: BTreeIndex<ProjectId,
                Option<(Project, ProjCommits)>>,
}
```

Structure 3b: Parts and Projects as Peers, Commits Subordinate to Projects

```
col!{ ProjCommits:
  RelProxy<Rc<
    BTreeIndex<PartId,
      Option<(PartId, QtyCommitted)>>>>
}

pub struct Inventory {
    parts:    BTreeIndex<PartId,
                Option<Part>>,
    projects: BTreeIndex<ProjectName,
                Option<(Project, ProjCommits)>>,
}
```

Structure 4a: Parts and Projects as Peers, Commits Subordinate to Parts

```
col!{ PartCommits:
  RelProxy<Rc<
    BTreeIndex<ProjectId,
      Option<(ProjectId, QtyCommitted)>>>>
}

pub struct Inventory {
    parts:    BTreeIndex<PartId,
                Option<(Part, PartCommits)>>,
    projects: BTreeIndex<ProjectId,
                Option<Project>>,
}
```

Structure 4b: Parts and Projects as Peers, Commits Subordinate to Parts

```
col!{ PartCommits:
  RelProxy<Rc<
    BTreeIndex<ProjectName,
      Option<(ProjectName,QtyCommitted)>>>>
}

pub struct Inventory {
    parts:    BTreeIndex<PartId,
                Option<(Part, PartCommits)>>,
    projects: BTreeIndex<ProjectName,
                Option<Project>>,
}
```

Structure 5a: Parts, Projects, and Commits as Peers

```
pub struct Inventory {
    parts:    BTreeIndex<PartId,
                Option<Part>>,
    projects: BTreeIndex<ProjectId,
                Option<Project>>,
    commits:  RedundantIndex<ProjectId, PartId,
                BTreeIndex<PartId,
                  Vec<Commit>>>,
}
```

Structure 5b: Parts, Projects, and Commits as Peers

```
pub struct Inventory {
    parts:    BTreeIndex<PartId,
                Option<Part>>,
    projects: RedundantIndex<ProjectName, ProjectId,
                BTreeIndex<ProjectId,
                  Option<Project>>>,
    commits: BTreeIndex<ProjectId,
               BTreeIndex<PartId,
                 Option<Commit>>>,
}
```

Appendix C: Rust Standard Library

This chapter contains brief descriptions of the types and traits from Rust's standard library which are referred to in this paper. For more detail, refer to the official documentation[9].

`&T`	type	A shared reference to a value of type `T`
`&mut T`	type	An exclusive reference to a value of type `T`
`Any`	trait	Used for downcasting from generic to concrete types
`Arc<T>`	type	A thread-safe reference with shared ownership
`AsRef<T>`	trait	Used to obtain a shared reference to a value of type `T`
`AsMut<T>`	trait	Used to obtain an exclusive reference to a value of type `T`
`bool`	type	A Boolean value
`Borrow<T>`	trait	Indicates that `Self` shares several properties with the type `T`, such as ordering, and can produce a shared reference to a value of type `T`
`BorrowMut<T>`	trait	Indicates that `Self` shares several properties with the type `T`, such as ordering, and can produce and exclusive reference to a value of type `T`
`Bound<T>`	type	An enumeration that describes one end of a range. Its value may be `Included(T)`, `Excluded(T)`, or `Unbounded`
`Box<T>`	type	An owned, heap-allocated value of type `T`
`BTreeSet<T>`	type	An ordered collection containing values of type `T`, with no duplicates
`Clone`	trait	Used to duplicate values
`Copy`	trait	Marker that indicates a bitwise copy is sufficient to duplicate values
`Default`	trait	Provides a constructor, `default()`, which takes no arguments
`Deref`	trait	Describes the behavior of the dereferencing operator (`*`) in immutable contexts.

9 https://doc.rust-lang.org/std/index.html

`DerefMut`	trait	Describeds the behavior of the dereferencing operator (`*`) in mutable contexts
`Eq`	trait	Defines the behavior of the equality operator (`==`)
`Fn`	trait	A closure that can be called via shared reference
`FnMut`	trait	A closure that can be called via exclusive reference
`FnOnce`	trait	A closure that can be called by consuming `Self`
`From<T>`	trait	Provides a constructor that takes a single argument of type `T`
`Into<T>`	trait	Provides a method to consume `Self` and produce a value of type `T`
`IntoIterator`	trait	Indicates that `Self` can be used to produce an iterator, usually implemented for collection types
`Iterator`	trait	Provides a mechanism to produce a sequence of values, one at a time
`Option<T>`	type	An enumeration that represents a possibly-missing value. Its value may be either `None` or `Some(T)`
`Ord`	trait	Indicates that values of type `Self` have a total ordering; defines the behavior of comparison operators
`Rc<T>`	type	A reference to a value of type `T` with shared ownership
`Result<T,E>`	type	An enumeration that represents the result of a fallible operation. Its value may be either `Ok(T)` or `Err(E)`
`Sized`	trait	A marker that indicates values of type `Self` have a compile-time known size
`String`	type	Owned UTF-8 encoded textual data, stored on the heap
`str`	type	UTF-8 encoded textual data, with length known only at runtime. Usually appears behind a shared reference: `&str`
`u32`	type	A 32-bit unsigned integer
`usize`	type	An unsigned integer with architecture-dependent width
`Vec<T>`	type	A growable collection containing values of type `T`, stored in a contiguous region of memory on the heap