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 (§...). 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 (§...). The set of column types contained in a record is known as its header. All column types also implement Record, as do tuples of records6. 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 (§...). This trait provides facilities to query the contained records, and the related traits Insert (§...) and Delete (§...) provide a common interface for modifying the contents of relations.

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

---

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

```rust
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:

```rust
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.

```rust
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:

```rust
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:

```rust
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.

```rust
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. 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:

```rust
#[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:

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

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, this 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<K2,K1,R> (§4.8.2) can be used to provide a secondary index on the PartName column:

```rust
#[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. Beside this, it stores a mapping from K2 (PartName) to K1 (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. 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:

```rust
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 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.

```rust
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:

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

```rust
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:

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

The QueryPlanImpl trait then describes how to construct a UniquePlan instance for a given relation and query:

```rust
impl<'a,C:Col+Ord,R,Q> QueryPlanImpl<'a,Unique<C,R>,Q> 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)]

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.