Common Lisp Relational Table Manager (RTM) Users Manual

Paul E. Silvey

CMPSCI Technical Report 90-86

Experimental Knowledge Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

RTM is a relational database management extension to Common Lisp. It provides capabilities for creating named tables with named column attributes of specified types (domains). Data access and manipulation is accomplished using a functional interface modeled after the database industry standard Structured Query Language (SQL).

This work was supported by University Research Initiative grant, ONR N00014-86-K-1764.

Chapter 1

Introduction

RTM is a relational database management extension to Common Lisp. It provides capabilities for creating named tables with named column attributes of specified types (domains). Data access and manipulation is accomplished using a functional interface modeled after the database industry standard Structured Query Language (SQL).

This Users Manual provides an overview of the relational database functionality of RTM, and a description of the programmer's interface. Chapter 2 discusses data definition, and Chapter 3 describes the data manipulation facilities. Chapter 4 discusses loading and initializing RTM in the Common Lisp environment.

While RTM is a relational database management tool, it does not provide some DBMS capabilites usually seen in commercial systems. For example, issues of concurrency and security have not been addressed at all, and database integrity issues are not handled in a complete way. Transaction processing and views are also not supported. However, because RTM is a database system for Common Lisp, it provides a degree of flexibility that commercial DBMS systems do not. Arbitrary Lisp data items can be stored in RTM tables, and with some care on the part of the user, they can be efficiently accessed. It is well suited as a rapid prototyping or single-user workstation tool.

A primary design goal of RTM was efficiency of ad-hoc access to large data sets in Lisp. In SQL, it is possible to write the same query in many different ways, and many of those ways lead to inefficient access, even with query optimization. A lot of time can be spent in allocating memory and copying data items, either as intermediate or final results. In a Lisp environment, allocating memory for temporary use means later having to garbage-collect it, and this can be quite wasteful. In RTM, the use of temporary data structures such as lists of attribute values is discouraged. Instead, iteration forms provide the functionality of SQL *cursors*, allowing access to table values without producing much garbage. Performing table joins as nested iterations adds a procedural feel to what is strictly declarative in SQL, but it is entirely natural to programmers and much more likely to be efficient. Some of the inherent power of the Relational Algebra or Relational Calculus has been thus restricted, mostly to force the programmer to take responsiblity for the efficiency of table joins.

Chapter 2

Data Definition

This chapter describes the facilities for defining a relational database schema in RTM.

2.1 Domains

Domains are the *data types* of RTM. All table attributes must be declared as belonging to either a built-in or user-defined domain. The domain declarations are necessary for database integrity. In particular, they are used for attribute validation during insert or update operations, and for defining attribute value orderings.

The following domains are provided by RTM:

- `fixnum`
- `integer`
- `float`
- `single-float`
- `double-float`
- `number`
- `string`
- `lowercase-string`
- `uppercase-string`
- `symbol`
- `date`
- `ugly`

Most users will probably not need to define their own domains, but to allow extensions, RTM provides this capability. The domain specific functions that the user defines will be called frequently, so for best results, they should be compiled functions that are as efficent as possible.

`create-domain` *name &key (less-p nil) (equal-p nil) (encode-fn nil) (decode-fn nil) (verify-fn nil) (default-value nil)* [*Function*]

Initializes RTM data structures to define an attribute domain.

A domain is specified by giving it a name, and by defining domain-specific functions for encoding, verifying, and decoding (*encode-fn*, *verify-fn*, *decode-fn*), functions for internal value matching and ordering (*equal-p* and *less-p*), and an internal default value for the domain, to be used when an attribute value is unknown or invalid (*defualt-value*).

When an attribute data value is inserted into an RTM table, it is first encoded by the *encode-fn*, if any. The encoded (internal) value is then verified by the *verify-fn*. If this predicate returns non `nil`, the internal value is inserted into the table. Otherwise, the domain default value is used. Whenever a value from an RTM table is returned to the user, the internal value is decoded using the *decode-fn*, if any.

The built-in numeric domains (`fixnum`, `integer`, `float`, `single-float`, `double-float`, and `number`) will coerce non-complex numeric values to the specified type during encoding, and return the internal value directly (i.e. there is no *decode-fn*). The default-values are equal to zero.

There are four built-in domains that represent their internal values as strings. They are `string`, `lowercase-string`, `uppercase-string`, and `symbol`. The domain `string` is case-sensitive, the others are not. Symbols are encoded as uppercase-strings for ordering, and to distinguish null values (default "") from Lisp `nil`.

The domain `date` encodes a "MM/DD/YY" date string into an integer (the number of days since the start of the 20th century). This domain is useful for verifying that dates are valid, and for ordering purposes. Values are decoded into the "MM/DD/YY" format.

The domain `ugly` is intended for storing Common Lisp data structures that are not atomic, such as lists, arrays, and structures. This is useful for associating this kind of data with other attributes that can be used as indexes to it. For example, you can store bitmaps as ugly data that are keyed by a numeric icon-id. This domain is the only built-in domain that is unordered. [1]

The following functions are provided for manual encoding and decoding of domain values.

`domain-encode-and-verify` *domain-name attr-value* [*Function*]

> Returns the internal form of attr-value after encoding and verifying it for specified domain. If attr-value is not valid for domain, a warning is generated and the default domain value is returned.

`domain-decode` *domain-name attr-value* [*Function*]

> Returns external form of attr-value after decoding it for specified domain.

RTM domains do not have to be atomic, as is prescribed by the pure relational database model. As long as they are properly defined, any Common Lisp data type can be stored in an RTM table. However, for efficient access to data elements, tables should adhere to the principles of good relational table design, namely the use of normal forms. Note also that loading and storing of arbitrary domains is not guaranteed to work properly.

[1] Warning: Ugly attribute values may not be Lisp printable / readable, so if stored to disk file, they may not load correctly (see the functions `store-table` and `load-table`).

2.2 Attributes and Tables

Relational tables are two dimensional arrays of data elements. Each data element in a table belongs to the domain of the attribute that names its column. When a table is created, its attributes are declared and remain fixed. Attributes can not be added or removed from a table without copying the data into another table of the desired structure. All data is accessed by name and/or value, never by its physical position in the table. In fact, the user can not actually look at the physical structure of the table - all access and manipulation of data is accomplished through the RTM interface functions and macros.

Attributes and tables are named by symbols. Strings may also work as names, but they are not recommended because they are less efficient and subject to case sensitivities. Attributes from two or more tables may share the same name, whether or not they belong to the same domain. Attributes do not have to belong to the same domain to be used as foreign keys, although this is also recommended as good practice.

A table can have a primary key, defined as the set of attributes specified with the *:key* qualifier when the table is created. On insert and update operations, RTM will verify that each row of the table has a unique value on those attributes when taken together.

RTM tables exist in the Common Lisp virtual memory space. Their persistence is achieved only through explicit storing to disk file and subsequently re-loading them. Since these operations are relatively slow, they should be used sparingly. A table can not be re-loaded if it currently exists - it should be dropped first. Indexes (discussed in the next section) are not written to disk files, but are created upon loading for attributes that had them at the time the table was stored.

The size of a table does not have to be declared, since tables grow in size as needed. However, for efficiency, it is better to specify as the `:initial-size` a liberal estimate of the total number of rows needed in the table. Table expansion, while automatic, involves copying data in memory. When a table does expand, it does so by a factor of `*rtm-table-expansion-factor*`, which is a system constant currently set to 1.5.

`create-table` *table-name &key (attributes nil) (initial-size *rtm-initial-table-size*)* [*Function*]

Creates table and returns t if successful or nil if error. Attributes are declared as a list of attribute specifiers, where each specifier consists of (attribute-name domain-name :key :index) where the optional keywords, when present, indicate that the attribute is part of the key field, and/or should be indexed, respectively. Domain names must have previously been declared, either by the system (built-in) or the user. (See documentation for create-domain for more information on built-in and user-defined domains.)

`drop-table` *table-name* [*Function*]

Removes specified table from system. Returns t is successful or nil if error.

`table-p` *table-name* [*Macro*]

Returns t if specified table exists, nil otherwise.

`table-attributes` *table-name* [*Function*]

Returns a list of table attribute names for specified table.

`create-view` *view-name &key (from nil) (where nil)* [*Macro*]

Create a view named view-name using specified :from table and :where clause query.

`drop-view` *view-name* [*Function*]

Drop the view named view-name.

`view-p` *table-name* [*Macro*]

Returns t if specified table is a view, nil otherwise.

`load-table` *table-name &key (static-p nil)* [*Function*]

Loads table rows from disk file and builds necessary indexes. If static-p is non-nil, then exact size will be allocated. Otherwise, table will expand by *rtm-table-expansion-factor*.

`store-table` *table-name* [*Function*]

Stores all rows of specified table or view to disk file in domain decoded attribute format.

2.3. Indexes

Table attributes in ordered domains can be indexed for more efficient access. An ordered domain is one that has *equal-p* and *less-p* predicates defined on it. When a table is created, attributes can be declared as indexed, but indexes can also be added or dropped at any time. Selections that could benefit from indexes when they are not available will generate warning messages. Indexes are implemented as B-tree data structures.

`create-index` *table-name attr-name* [*Function*]

Creates a B-tree index on specified attribute of table. Returns t if successful or nil if error.

`drop-index` *table-name attr-name* [*Function*]

Removes the B-tree index on specified attribute of table. Returns t if successful or nil if error.

`index-p` *table-name attr-name* [*Function*]

Returns t if index exists for specified table and attribute, nil otherwise.

Chapter 3

Data Manipulation

This chapter describes the facilities for inserting, accessing, updating, and deleting rows in RTM tables. The data manipulation language is modeled after SQL - in particular, the selection forms use a keyword syntax for the **FROM**, **WHERE**, and **ORDER-BY** clauses of SQL.

3.1 Inserting Data

Data is inserted into tables one row at a time. Attributes that are not specified on insert will use their domain default values. If the table has a primary key, the row will only be inserted if it is unique on that key (set of attributes).

`insert-into-table` *table-name attr-value-list* [*Function*]

> Inserts table row using specified attribute values or domain defaults, updates necessary indexes, and returns t if successful or nil if error. Attribute values are specified using property list syntax, that is, as a sequence of pairs of attribute name followed by attribute value. For example, (insert-into-table table-name (list a1-name a1-value a2-name a2-value)).

3.2 Accessing and Updating Data

The basic query mechanism in RTM utilizes the `list-selection` or `do-selection` forms. Selection forms take a single table name as the `:from` keyword argument (or `:in-table` for `do-neighbors`). Joins of multiple tables are accomplished through nesting of selections. The `:where` keyword argument specifies the query selection criteria (analogous to the restriction or selection operator of Relational Algebra). The result of a selection is the retrieval of, or iteration over, the subset of table rows that match the where clause specification.

Access to specific attribute values in the selected table rows is achieved through the use of the `:attrs` keyword argument for `list-selection`, or through the use of the `attr-value` form from within the body of a `do-selection` or `do-neighbors` form. These iteration constructs

are analogous to SQL *cursors*. An attribute value is updated by a `setf` of `attr-value`.

3.2.1 Primary Selection Forms

`List-selction` and `do-selection` also take an optional `:order-by` keyword argument. The syntax for an order-by clause is a list of table attribute names preceded by the optional direction qualifiers `:asc` or `:desc` for ascending and descending order, respectively. The default ordering is `:asc`, and once a qualifier has been specified in the list, it remains in effect for subsequent attribute names, unless specificially changed.

`list-selection` *&key (attrs nil) from (where nil) (order-by nil)* [*Macro*]

Returns a list of values if attrs is a single attr-name, or a list of lists (rows of projected attribute values) if attrs is a list of attribute names.

`do-selection` *(&key (from nil) (where nil) (order-by nil)) &body body* [*Macro*]

Iterates over selected table rows. Projection of attributes (access to values) is achieved through the use of (attr-value attr-name) from within body of form. Attribute values may be modified using (setf (attr-value attr-name) attr-value). Returns value of last form executed in loop or nil if error.

`attr-value` *attr-name* [*Macro*]

Accesses table attribute values in selection context. Only valid within the lexical scope of a do-selection or do-neighbors form. To update table attribute values, use setf of this form.

`row-exists-p` *&key in-table (where nil)* [*Macro*]

Returns t if any table row satisfies specified selection criteria, nil otherwise.

3.2.2 Selection Where Clause Forms

The selection forms all take a where clause, which is a logical composition of attribute comparison operators. The operators are macros that should be used only in the context of the `:where` keyword argument in selection forms. If a where clause is not specified or is `nil`, it will match all table rows.

Where Clause Logic Operators

The following macros can be used to create logical combinations of attribute comparisons in selection where clauses.

`.and.` *arg1 &rest args* [*Macro*]

Selects rows matching every argument's selection criteria.

`.or.` *arg1 &rest args* [*Macro*]

Selects rows matching at least one argument's selection criteria.

`.not.` *arg* [*Macro*]

Selects rows not matching argument's selection criteria.

Where Clause Comparison Operators

The following macros can be used for attribute comparison in selection where clauses.

.==. *attr-name attr-value* [*Macro*]

Selects rows where table attribute is equal to query attribute.

.<=. *attr-name attr-value* [*Macro*]

Selects rows where table attribute is less than or equal to query attribute.

.>=. *attr-name attr-value* [*Macro*]

Selects rows where table attribute is greater than or equal to query attribute.

./=. *attr-name attr-value* [*Macro*]

Selects rows where table attribute is not equal to query attribute.

.>>. *attr-name attr-value* [*Macro*]

Selects rows where table attribute is greater than query attribute.

.<<. *attr-name attr-value* [*Macro*]

Selects rows where table attribute is less than query attribute.

.=><=. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is greater than or equal to first query attribute and less than or equal to second query attribute.

.=><<. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is greater than or equal to first query attribute and less than second query attribute.

.>><=. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is greater than first query attribute and less than or equal to second query attribute.

.>><<. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is greater than first query attribute and less than second query attribute.

.<<>>. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is less than first query attribute and greater than second query attribute.

.<<=>. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is less than first query attribute and greater than or equal to second query attribute.

.<=>>. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is less than or equal to first query attribute and greater than second query attribute.

.<==>. *attr-name attr-value-1 attr-value-2* [*Macro*]

Selects rows where table attribute is less than or equal to first query attribute and greater than or equal to second query attribute.

`.min.` *attr-name* [*Macro*]

Selects rows where table attribute is the minimum value in table.

`.max.` *attr-name* [*Macro*]

Selects rows where table attribute is the maximum value in table.

`.pred.` *attr-name attr-value* [*Macro*]

Selects rows where table attribute has greatest value less than query attribute.

`.succ.` *attr-name attr-value* [*Macro*]

Selects rows where table attribute has smallest value greater than query attribute.

`.floor.` *attr-name attr-value* [*Macro*]

Selects rows where table attribute has greatest value less than or equal to query attribute.

`.ceiling.` *attr-name attr-value* [*Macro*]

Selects rows where table attribute has smallest value greater than or equal to query attribute.

3.2.3 Aggregate and Special Selection Forms

Aggregate operators return a single value that represents some collective characteristic of the set of rows that match the query where clause. The following aggregate query macros are currently supported. They are implemented more efficiently than could be done using only the selection forms above.

`count-selection` *&key from (where nil)* [*Macro*]

Returns count of number of table rows that match selection criteria.

`min-selection` *&key attr from (where nil)* [*Macro*]

Returns the minimum value of specified attribute over rows matching selection criteria.

`max-selection` *&key attr from (where nil)* [*Macro*]

Returns the maximum value of specified attribute over rows matching selection criteria.

The `do-neighbors` form is a fast way to do nearest-neighbor searches in n-dimensional numeric attribute spaces. It has an advantage over window-based searches (.e.g. using the range comparison operators) in that it allows arbitrary neighborhood sizes. The stopping criteria can therefore be data dependent. The distance functions assume the domains are numeric.

`do-neighbors` *(distance &key (from-point nil) (in-table nil) (distance-metric nil))* [*Macro*]
&body body

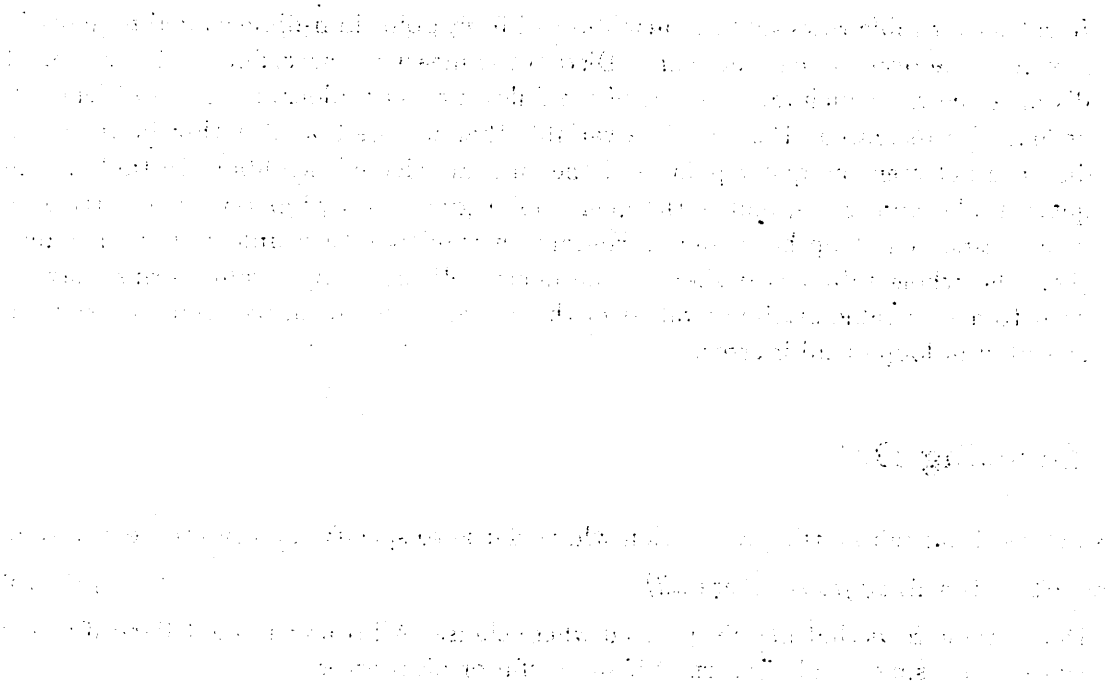
Iterates over table rows that are neighbors of from-point in n-dimensional minkowski metric space over numeric domains. Distance is measured according to the specified distance-metric, which must be one of the following: :city-block, :euclidean (default), or :euclidean-squared. Distance is a variable that will be bound within body to the distance between the query point and the current selected neighbor. In-table is the query table, and from-point is the query point given as a property list of attribute value pairs. The loop body should contain an explicit return unless every neighbor (i.e. the whole table) is desired in the query selection. Attr-value forms may be used to access table attribute values of the current row. Returns value of last form executed in loop or nil if error.

3.3 Deleting Data

Data is deleted from tables using a selection where clause to specify the rows to be removed.

`delete-selection` *&key from (where nil)* [*Macro*]

Delete table rows that match specified where clause. All rows will be deleted if where clause is missing or nil. Returns t if successful or nil if error.



Chapter 4

Initializing RTM

???

In order to use RTM, you must first load the system code. To do this, first load the file "*init-fns*". Tell Common Lisp that you will be using the RTM package using use-package, and then evaluate the function `load-rtm`. The system is initialized upon loading, but if you wish to restart it, use the function `rtm-reset`.

You should specify a directory pathname where you want to store your RTM tables. This is done by binding `*rtm-data-directory*`.

There are three typeout streams that RTM uses to communicate information to the user, `*rtm-error-stream*`, `*rtm-warning-stream*`, and `*rtm-notification-stream*`. Errors will cause a break; they probably indicate an RTM bug. Warnings are continuable, and the forms that caused them should return `nil`. Notifications are for providing information to the user on disk file activity.

`*rtm-initial-table-size*` [*Variable*]

Default initial table size.

`*rtm-table-expansion-factor*` [*Constant*]

Expansion factor for automatic table size adjustment.

`*rtm-data-directory*` [*Variable*]

Disk file system directory where RTM tables are stored.

`*rtm-error-stream*` [*Variable*]

Stream where RTM fatal error messages are sent.

`*rtm-warning-stream*` [*Variable*]

Stream where RTM warning messages are sent.

`*rtm-notification-stream*` [*Variable*]

Stream where RTM notifications are sent.

`load-rtm` *&key (pkg user)* [*Function*]

Load Relational Table Manager system for use in specified package.

`rtm-reset` *&key (pkg user) (cleanup t)* [*Function*]

Reset Relational Table Manager (RTM) system. Initialize RTM default domains and intern as symbols in specified package. If cleanup requested, store any changed tables.

`rtm-cleanup` [*Function*]

Store all changed tables to disk.

Index