27. Practical: An MP3 Database

In this chapter you'll revisit the idea first explored in Chapter 3 of building an in-memory database out of basic Lisp data structures. This time your goal is to hold information that you'll extract from a collection of MP3 files using the ID3v2 library from Chapter 25. You'll then use this database in Chapters 28 and 29 as part of a Web-based streaming MP3 server. Of course, this time around you can use some of the language features you've learned since Chapter 3 to build a more sophisticated version.

The Database

The main problem with the database in Chapter 3 is that there's only one table, the list stored in the variable `*db*`. Another is that the code doesn't know anything about what type of values are stored in different columns. In Chapter 3 you got away with that by using the fairly general-purpose **`EQUAL`** method to compare column values when selecting rows from the database, but you would've been in trouble if you had wanted to store values that couldn't be compared with **`EQUAL`** or if you had wanted to sort the rows in the database since there's no ordering function that's as general as **`EQUAL`**.

This time you'll solve both problems by defining a class, `table`, to represent individual database tables. Each `table` instance will consist of two slots--one to hold the table's data and another to hold information about the columns in the table that database operations will be able to use. The class looks like this:

```
(defclass table ()
  ((rows   :accessor rows   :initarg :rows :initform (make-rows))
   (schema :accessor schema :initarg :schema)))
```

As in Chapter 3, you can represent the individual rows with plists, but this time around you'll create an abstraction that will make that an implementation detail you can change later without too much trouble. And this time you'll store the rows in a vector rather than a list since certain operations that you'll want to support, such as random access to rows by a numeric index and the ability to sort a table, can be more efficiently implemented with vectors.

The function `make-rows` used to initialize the `rows` slot can be a simple wrapper around **`MAKE-ARRAY`** that builds an empty, adjustable,vector with a fill pointer.

The Package

The package for the code you'll develop in this chapter looks like this:

```
(defpackage :com.gigamonkeys.mp3-database
  (:use :common-lisp
        :com.gigamonkeys.pathnames
        :com.gigamonkeys.macro-utilities
```

```
            :com.gigamonkeys.id3v2)
      (:export  :*default-table-size*
                :*mp3-schema*
                :*mp3s*
                :column
                :column-value
                :delete-all-rows
                :delete-rows
                :do-rows
                :extract-schema
                :in
                :insert-row
                :load-database
                :make-column
                :make-schema
                :map-rows
                :matching
                :not-nullable
                :nth-row
                :random-selection
                :schema
                :select
                :shuffle-table
                :sort-rows
                :table
                :table-size
                :with-column-values))
```

The `:use` section gives you access to the functions and macros whose names are exported from the packages defined in Chapter 15, 8, and 25 and the `:export` section exports the API this library will provide, which you'll use in Chapter 29.

```
(defparameter *default-table-size* 100)

(defun make-rows (&optional (size *default-table-size*))
  (make-array size :adjustable t :fill-pointer 0))
```

To represent a table's schema, you need to define another class, `column`, each instance of which will contain information about one column in the table: its name, how to compare values in the column for equality and ordering, a default value, and a function that will be used to normalize the column's values when inserting data into the table and when querying the table. The `schema` slot will hold a list of `column` objects. The class definition looks like this:

```
(defclass column ()
  ((name
    :reader name
    :initarg :name)

   (equality-predicate
    :reader equality-predicate
    :initarg :equality-predicate)

   (comparator
    :reader comparator
    :initarg :comparator)

   (default-value
    :reader default-value
    :initarg :default-value
    :initform nil)

   (value-normalizer
    :reader value-normalizer
    :initarg :value-normalizer
    :initform #'(lambda (v column) (declare (ignore column)) v))))
```

The `equality-predicate` and `comparator` slots of a `column` object hold functions used to compare values from the given column for equivalence and ordering. Thus, a column

containing string values might have **STRING=** as its `equality-predicate` and **STRING<** as its `comparator`, while a column containing numbers might have **=** and **<**.

The `default-value` and `value-normalizer` slots are used when inserting rows into the database and, in the case of `value-normalizer`, when querying the database. When you insert a row into the database, if no value is provided for a particular column, you can use the value stored in the `column`'s `default-value` slot. Then the value--defaulted or otherwise--is normalized by passing it and the column object to the function stored in the `value-normalizer` slot. You pass the column in case the `value-normalizer` function needs to use some data associated with the column object. (You'll see an example of this in the next section.) You should also normalize values passed in queries before comparing them with values in the database.

Thus, the `value-normalizer`'s responsibility is primarily to return a value that can be safely and correctly passed to the `equality-predicate` and `comparator` functions. If the `value-normalizer` can't figure out an appropriate value to return, it can signal an error.

The other reason to normalize values before you store them in the database is to save both memory and CPU cycles. For instance, if you have a column that's going to contain string values but the number of distinct strings that will be stored in the column is small--for instance, the genre column in the MP3 database--you can save space and speed by using the `value-normalizer` to *intern* the strings (translate all **STRING=** values to a single string object). Thus, you'll need only as many strings as there are distinct values, regardless of how many rows are in the table, and you can use **EQL** to compare column values rather than the slower **STRING=**.[1]

Defining a Schema

Thus, to make an instance of `table`, you need to build a list of `column` objects. You could build the list by hand, using **LIST** and **MAKE-INSTANCE**. But you'll soon notice that you're frequently making a lot column objects with the same comparator and equality-predicate combinations. This is because the combination of a comparator and equality predicate essentially defines a column type. It'd be nice if there was a way to give those types names that would allow you to say simply that a given column is a string column, rather than having to specify **STRING<** as its comparator and **STRING=** as its equality predicate. One way is to define a generic function, `make-column`, like this:

```
(defgeneric make-column (name type &optional default-value))
```

Now you can implement methods on this generic function that specialize on `type` with **EQL** specializers and return `column` objects with the slots filled in with appropriate values. Here's the generic function and methods that define column types for the type names `string` and `number`:

```
(defmethod make-column (name (type (eql 'string)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'string<
```

```
   :equality-predicate #'string=
   :default-value default-value
   :value-normalizer #'not-nullable))

(defmethod make-column (name (type (eql 'number)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'<
   :equality-predicate #'=
   :default-value default-value))
```

The following function, `not-nullable`, used as the `value-normalizer` for `string` columns, simply returns the value it's given unless the value is **`NIL`**, in which case it signals an error:

```
(defun not-nullable (value column)
  (or value (error "Column ~a can't be null" (name column))))
```

This is important because **`STRING<`** and **`STRING=`** will signal an error if called on **`NIL`**; it's better to catch bad values before they go into the table rather than when you try to use them.[2]

Another column type you'll need for the MP3 database is an `interned-string` whose values are interned as discussed previously. Since you need a hash table in which to intern values, you should define a subclass of `column`, `interned-values-column`, that adds a slot whose value is the hash table you use to intern.

To implement the actual interning, you'll also need to provide an `:initform` for `value-normalizer` of a function that interns the value in the column's `interned-values` hash table. And because one of the main reasons to intern values is to allow you to use **`EQL`** as the equality predicate, you should also add an `:initform` for the `equality-predicate` of `#'eql`.

```
(defclass interned-values-column (column)
  ((interned-values
    :reader interned-values
    :initform (make-hash-table :test #'equal))
   (equality-predicate :initform #'eql)
   (value-normalizer   :initform #'intern-for-column)))

(defun intern-for-column (value column)
  (let ((hash (interned-values column)))
    (or (gethash (not-nullable value column) hash)
        (setf (gethash value hash) value))))
```

You can then define a `make-column` method specialized on the name `interned-string` that returns an instance of `interned-values-column`.

```
(defmethod make-column (name (type (eql 'interned-string)) &optional default-value)
  (make-instance
   'interned-values-column
   :name name
   :comparator #'string<
   :default-value default-value))
```

With these methods defined on `make-column`, you can now define a function, `make-schema`, that builds a list of `column` objects from a list of column specifications consisting of a column name, a column type name, and, optionally, a default value.

```
(defun make-schema (spec)
  (mapcar #'(lambda (column-spec) (apply #'make-column column-spec)) spec))
```

For instance, you can define the schema for the table you'll use to store data extracted from MP3s like this:

```
(defparameter *mp3-schema*
  (make-schema
   '((:file     string)
     (:genre    interned-string "Unknown")
     (:artist   interned-string "Unknown")
     (:album    interned-string "Unknown")
     (:song     string)
     (:track    number 0)
     (:year     number 0)
     (:id3-size number))))
```

To make an actual table for holding information about MP3s, you pass `*mp3-schema*` as the `:schema` initarg to **`MAKE-INSTANCE`**.

```
(defparameter *mp3s* (make-instance 'table :schema *mp3-schema*))
```

Inserting Values

Now you're ready to define your first table operation, `insert-row`, which takes a plist of names and values and a table and adds a row to the table containing the given values. The bulk of the work is done in a helper function, `normalize-row`, that builds a plist with a defaulted, normalized value for each column, using the values from `names-and-values` if available and the `default-value` for the column if not.

```
(defun insert-row (names-and-values table)
  (vector-push-extend (normalize-row names-and-values (schema table)) (rows table)))

(defun normalize-row (names-and-values schema)
  (loop
     for column in schema
     for name  = (name column)
     for value = (or (getf names-and-values name) (default-value column))
     collect name
     collect (normalize-for-column value column)))
```

It's worth defining a separate helper function, `normalize-for-column`, that takes a value and a `column` object and returns the normalized value because you'll need to perform the same normalization on query arguments.

```
(defun normalize-for-column (value column)
  (funcall (value-normalizer column) value column))
```

Now you're ready to combine this database code with code from previous chapters to build a database of data extracted from MP3 files. You can define a function, `file->row`, that uses `read-id3` from the ID3v2 library to extract an ID3 tag from a file and turns it into a plist that you can pass to `insert-row`.

```
(defun file->row (file)
  (let ((id3 (read-id3 file)))
    (list
     :file   (namestring (truename file))
     :genre  (translated-genre id3)
     :artist (artist id3)
     :album  (album id3)
     :song   (song id3)
     :track  (parse-track (track id3))
     :year   (parse-year (year id3))
     :id3-size (size id3))))
```

You don't have to worry about normalizing the values since `insert-row` takes care of that for you. You do, however, have to convert the string values returned by the `track` and `year` into numbers. The track number in an ID3 tag is sometimes stored as the ASCII representation of the track number and sometimes as a number followed by a slash followed by the total number of tracks on the album. Since you care only about the actual track number, you should use the `:end` argument to **`PARSE-INTEGER`** to specify that it should parse only up to the slash, if any.[3]

```
(defun parse-track (track)
  (when track (parse-integer track :end (position #\/ track))))

(defun parse-year (year)
  (when year (parse-integer year)))
```

Finally, you can put all these functions together, along with `walk-directory` from the portable pathnames library and `mp3-p` from the ID3v2 library, to define a function that loads an MP3 database with data extracted from all the MP3 files it can find under a given directory.

```
(defun load-database (dir db)
  (let ((count 0))
    (walk-directory
     dir
     #'(lambda (file)
         (princ #\.)
         (incf count)
         (insert-row (file->row file) db))
     :test #'mp3-p)
    (format t "~&Loaded ~d files into database." count)))
```

Querying the Database

Once you've loaded your database with data, you'll need a way to query it. For the MP3 application you'll need a slightly more sophisticated query function than you wrote in Chapter 3. This time around you want not only to be able to select rows matching particular criteria but also to limit the results to particular columns, to limit the results to unique rows, and perhaps to sort the rows by particular columns. In keeping with the spirit of relational database theory, the result of a query will be a new `table` object containing the desired rows and columns.

The query function you'll write, `select`, is loosely modeled on the `SELECT` statement from Structured Query Language (SQL). It'll take five keyword parameters: `:from`, `:columns`, `:where`, `:distinct`, and `:order-by`. The `:from` argument is the `table` object you want to query. The `:columns` argument specifies which columns should be included in the result. The value should be a list of column names, a single column name, or a **`T`**, the default, meaning return all columns. The `:where` argument, if provided, should be a function that accepts a row and returns true if it should be included in the results. In a moment, you'll write two functions, `matching` and `in`, that return functions appropriate for use as `:where` arguments. The `:order-by` argument, if supplied, should be a list of column names; the results will be sorted by the named columns. As with the `:columns` argument, you can specify a single column using just the name, which is equivalent to a one-item list containing the same name. Finally, the `:distinct` argument is a boolean that says whether to eliminate duplicate rows from the results. The default value for `:distinct` is **`NIL`**.

Here are some examples of using `select`:

```
;; Select all rows where the :artist column is "Green Day"
(select :from *mp3s* :where (matching *mp3s* :artist "Green Day"))

;; Select a sorted list of artists with songs in the genre "Rock"
(select
  :columns :artist
  :from *mp3s*
  :where (matching *mp3s* :genre "Rock")
  :distinct t
  :order-by :artist)
```

The implementation of `select` with its immediate helper functions looks like this:

```
(defun select (&key (columns t) from where distinct order-by)
  (let ((rows (rows from))
        (schema (schema from)))

    (when where
      (setf rows (restrict-rows rows where)))

    (unless (eql columns 't)
      (setf schema (extract-schema (mklist columns) schema))
      (setf rows (project-columns rows schema)))

    (when distinct
      (setf rows (distinct-rows rows schema)))

    (when order-by
      (setf rows (sorted-rows rows schema (mklist order-by))))

    (make-instance 'table :rows rows :schema schema)))

(defun mklist (thing)
  (if (listp thing) thing (list thing)))

(defun extract-schema (column-names schema)
  (loop for c in column-names collect (find-column c schema)))

(defun find-column (column-name schema)
  (or (find column-name schema :key #'name)
      (error "No column: ~a in schema: ~a" column-name schema)))

(defun restrict-rows (rows where)
  (remove-if-not where rows))

(defun project-columns (rows schema)
  (map 'vector (extractor schema) rows))

(defun distinct-rows (rows schema)
  (remove-duplicates rows :test (row-equality-tester schema)))

(defun sorted-rows (rows schema order-by)
  (sort (copy-seq rows) (row-comparator order-by schema)))
```

Of course, the really interesting part of `select` is how you implement the functions `extractor`, `row-equality-tester`, and `row-comparator`.

As you can tell by how they're used, each of these functions must return a function. For instance, `project-columns` uses the value returned by `extractor` as the function argument to **`MAP`**. Since the purpose of `project-columns` is to return a set of rows with only certain column values, you can infer that `extractor` returns a function that takes a row as an argument and returns a new row containing only the columns specified in the schema it's passed. Here's how you can implement it:

```
(defun extractor (schema)
  (let ((names (mapcar #'name schema)))
    #'(lambda (row)
        (loop for c in names collect c collect (getf row c)))))
```

Note how you can do the work of extracting the names from the schema outside the body of the closure: since the closure will be called many times, you want it to do as little work as possible each time it's called.

The functions `row-equality-tester` and `row-comparator` are implemented in a similar way. To decide whether two rows are equivalent, you need to apply the appropriate equality predicate for each column to the appropriate column values. Recall from Chapter 22 that the **`LOOP`** clause `always` will return **`NIL`** as soon as a pair of values fails their test or will cause the **`LOOP`** to return **`T`**.

```
(defun row-equality-tester (schema)
  (let ((names (mapcar #'name schema))
        (tests (mapcar #'equality-predicate schema)))
    #'(lambda (a b)
        (loop for name in names and test in tests
           always (funcall test (getf a name) (getf b name))))))
```

Ordering two rows is a bit more complex. In Lisp, comparator functions return true if their first argument should be sorted ahead of the second and **`NIL`** otherwise. Thus, a **`NIL`** can mean that the second argument should be sorted ahead of the first *or* that they're equivalent. You want your row comparators to behave the same way: return **`T`** if the first row should be sorted ahead of the second and **`NIL`** otherwise.

Thus, to compare two rows, you should compare the values from the columns you're sorting by, in order, using the appropriate comparator for each column. First call the comparator with the value from the first row as the first argument. If the comparator returns true, that means the first row should definitely be sorted ahead of the second row, so you can immediately return **`T`**.

But if the column comparator returns **`NIL`**, then you need to determine whether that's because the second value should sort ahead of the first value or because they're equivalent. So you should call the comparator again with the arguments reversed. If the comparator returns true this time, it means the second column value sorts ahead of the first and thus the second row ahead of the first row, so you can return **`NIL`** immediately. Otherwise, the column values are equivalent, and you need to move onto the next column. If you get through all the columns without one row's value ever winning the comparison, then the rows are equivalent, and you return **`NIL`**. A function that implements this algorithm looks like this:

```
(defun row-comparator (column-names schema)
  (let ((comparators (mapcar #'comparator (extract-schema column-names schema))))
    #'(lambda (a b)
        (loop
           for name in column-names
           for comparator in comparators
           for a-value = (getf a name)
           for b-value = (getf b name)
           when (funcall comparator a-value b-value) return t
           when (funcall comparator b-value a-value) return nil
           finally (return nil)))))
```

Matching Functions

The `:where` argument to `select` can be any function that takes a row object and returns true if it should be included in the results. In practice, however, you'll rarely need the full power of arbitrary code to express query criteria. So you should provide two functions, `matching` and

in, that will build query functions that allow you to express the common kinds of queries and that take care of using the proper equality predicates and value normalizers for each column.

The workhouse query-function constructor will be matching, which returns a function that will match rows with specific column values. You saw how it was used in the earlier examples of select. For instance, this call to matching:

```
(matching *mp3s* :artist "Green Day")
```

returns a function that matches rows whose :artist value is "Green Day". You can also pass multiple names and values; the returned function matches when all the columns match. For example, the following returns a closure that matches rows where the artist is "Green Day" and the album is "American Idiot":

```
(matching *mp3s* :artist "Green Day" :album "American Idiot")
```

You have to pass matching the table object because it needs access to the table's schema in order to get at the equality predicates and value normalizer functions for the columns it matches against.

You build up the function returned by matching out of smaller functions, each responsible for matching one column's value. To build these functions, you should define a function, column-matcher, that takes a column object and an unnormalized value you want to match and returns a function that accepts a single row and returns true when the value of the given column in the row matches the normalized version of the given value.

```
(defun column-matcher (column value)
  (let ((name (name column))
        (predicate (equality-predicate column))
        (normalized (normalize-for-column value column)))
    #'(lambda (row) (funcall predicate (getf row name) normalized))))
```

You then build a list of column-matching functions for the names and values you care about with the following function, column-matchers:

```
(defun column-matchers (schema names-and-values)
  (loop for (name value) on names-and-values by #'cddr
     when value collect
       (column-matcher (find-column name schema) value)))
```

Now you can implement matching. Again, note that you do as much work as possible outside the closure in order to do it only once rather than once per row in the table.

```
(defun matching (table &rest names-and-values)
  "Build a where function that matches rows with the given column values."
  (let ((matchers (column-matchers (schema table) names-and-values)))
    #'(lambda (row)
        (every #'(lambda (matcher) (funcall matcher row)) matchers))))
```

This function is a bit of a twisty maze of closures, but it's worth contemplating for a moment to get a flavor of the possibilities of programming with functions as first-class objects.

The job of matching is to return a function that will be invoked on each row in a table to determine whether it should be included in the new table. So, matching returns a closure with one parameter, row.

Now recall that the function **EVERY** takes a predicate function as its first argument and returns true if, and only if, that function returns true each time it's applied to an element of the list passed as **EVERY**'s second argument. However, in this case, the list you pass to **EVERY** is itself a list of functions, the column matchers. What you want to know is that every column matcher, when invoked on the row you're currently testing, returns true. So, as the predicate argument to **EVERY**, you pass yet another closure that **FUNCALL**s the column matcher, passing it the row.

Another matching function that you'll occasionally find useful is `in`, which returns a function that matches rows where a particular column is in a given set of values. You'll define `in` to take two arguments: a column name and a table that contains the values you want to match. For instance, suppose you wanted to find all the songs in the MP3 database that have names the same as a song performed by the Dixie Chicks. You can write that where clause using `in` and a sub`select` like this:[4]

```
(select
  :columns '(:artist :song)
  :from *mp3s*
  :where (in :song
             (select
               :columns :song
               :from *mp3s*
               :where (matching *mp3s* :artist "Dixie Chicks"))))
```

Although the queries are more complex, the definition of `in` is much simpler than that of `matching`.

```
(defun in (column-name table)
  (let ((test (equality-predicate (find-column column-name (schema table))))
        (values (map 'list #'(lambda (r) (getf r column-name)) (rows table))))
    #'(lambda (row)
        (member (getf row column-name) values :test test))))
```

Getting at the Results

Since `select` returns another `table`, you need to think a bit about how you want to get at the individual row and column values in a table. If you're sure you'll never want to change the way you represent the data in a table, you can just make the structure of a table part of the API--that `table` has a slot `rows` that's a vector of plists--and use all the normal Common Lisp functions for manipulating vectors and plists to get at the values in the table. But that representation is really an internal detail that you might want to change. Also, you don't necessarily want other code manipulating the data structures directly--for instance, you don't want anyone to use **SETF** to put an unnormalized column value into a row. So it might be a good idea to define a few abstractions that provide the operations you want to support. Then if you decide to change the internal representation later, you'll need to change only the implementation of these functions and macros. And while Common Lisp doesn't enable you to absolutely prevent folks from getting at "internal" data, by providing an official API you at least make it clear where the boundary is.

Probably the most common thing you'll need to do with the results of a query is to iterate over the individual rows and extract specific column values. So you need to provide a way to do both those things without touching the `rows` vector directly or using **GETF** to get at the column values within a row.

For now these operations are trivial to implement; they're merely wrappers around the code you'd write if you didn't have these abstractions. You can provide two ways to iterate over the rows of a table: a macro `do-rows`, which provides a basic looping construct, and a function `map-rows`, which builds a list containing the results of applying a function to each row in the table.[5]

```
(defmacro do-rows ((row table) &body body)
  `(loop for ,row across (rows ,table) do ,@body))

(defun map-rows (fn table)
  (loop for row across (rows table) collect (funcall fn row)))
```

To get at individual column values within a row, you should provide a function, `column-value`, that takes a row and a column name and returns the appropriate value. Again, it's a trivial wrapper around the code you'd write otherwise. But if you change the internal representation of a table later, users of `column-value` needn't be any the wiser.

```
(defun column-value (row column-name)
  (getf row column-name))
```

While `column-value` is a sufficient abstraction for getting at column values, you'll often want to get at the values of multiple columns at once. So you can provide a bit of syntactic sugar, a macro, `with-column-values`, that binds a set of variables to the values extracted from a row using the corresponding keyword names. Thus, instead of writing this:

```
(do-rows (row table)
  (let ((song (column-value row :song))
        (artist (column-value row :artist))
        (album (column-value row :album)))
    (format t "~a by ~a from ~a~%" song artist album)))
```

you can simply write the following:

```
(do-rows (row table)
  (with-column-values (song artist album) row
    (format t "~a by ~a from ~a~%" song artist album)))
```

Again, the actual implementation isn't complicated if you use the `once-only` macro from Chapter 8.

```
(defmacro with-column-values ((&rest vars) row &body body)
  (once-only (row)
    `(let ,(column-bindings vars row) ,@body)))

(defun column-bindings (vars row)
  (loop for v in vars collect `(,v (column-value ,row ,(as-keyword v)))))

(defun as-keyword (symbol)
  (intern (symbol-name symbol) :keyword))
```

Finally, you should provide abstractions for getting at the number of rows in a table and for accessing a specific row by numeric index.

```
(defun table-size (table)
  (length (rows table)))

(defun nth-row (n table)
  (aref (rows table) n))
```

Other Database Operations

Finally, you'll implement a few other database operations that you'll need in Chapter 29. The first two are analogs of the SQL `DELETE` statement. The function `delete-rows` is used to delete rows from a table that match particular criteria. Like `select`, it takes `:from` and `:where` keyword arguments. Unlike `select`, it doesn't return a new table--it actually modifies the table passed as the `:from` argument.

```
(defun delete-rows (&key from where)
  (loop
     with rows = (rows from)
     with store-idx = 0
     for read-idx from 0
     for row across rows
     do (setf (aref rows read-idx) nil)
     unless (funcall where row) do
       (setf (aref rows store-idx) row)
       (incf store-idx)
     finally (setf (fill-pointer rows) store-idx)))
```

In the interest of efficiency, you might want to provide a separate function for deleting all the rows from a table.

```
(defun delete-all-rows (table)
  (setf (rows table) (make-rows *default-table-size*)))
```

The remaining table operations don't really map to normal relational database operations but will be useful in the MP3 browser application. The first is a function to sort the rows of a table in place.

```
(defun sort-rows (table &rest column-names)
  (setf (rows table) (sort (rows table) (row-comparator column-names (schema table))))
  table)
```

On the flip side, in the MP3 browser application, you'll need a function that shuffles a table's rows in place using the function `nshuffle-vector` from Chapter 23.

```
(defun shuffle-table (table)
  (nshuffle-vector (rows table))
  table)
```

And finally, again for the purposes of the MP3 browser, you should provide a function that selects *n* random rows, returning the results as a new table. It also uses `nshuffle-vector` along with a version of `random-sample` based on Algorithm S from Donald Knuth's *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Third Edition (Addison-Wesley, 1998) that I discussed in Chapter 20.

```
(defun random-selection (table n)
  (make-instance
   'table
   :schema (schema table)
   :rows (nshuffle-vector (random-sample (rows table) n))))

(defun random-sample (vector n)
  "Based on Algorithm S from Knuth. TAOCP, vol. 2. p. 142"
  (loop with selected = (make-array n :fill-pointer 0)
     for idx from 0
     do
       (loop
          with to-select = (- n (length selected))
          for remaining = (- (length vector) idx)
          while (>= (* remaining (random 1.0)) to-select)
          do (incf idx))
       (vector-push (aref vector idx) selected)
     when (= (length selected) n) return selected))
```

With this code you'll be ready, in Chapter 29, to build a Web interface for browsing a collection of MP3 files. But before you get to that, you need to implement the part of the server that streams MP3s using the Shoutcast protocol, which is the topic of the next chapter.

---

[1]The general theory behind interning objects is that if you're going to compare a particular value many times, it's worth it to pay the cost of interning it. The `value-normalizer` runs once when you insert a value into the table and, as you'll see, once at the beginning of each query. Since a query can involve invoking the `equality-predicate` once per row in the table, the amortized cost of interning the values will quickly approach zero.

[2]As always, the first causality of concise exposition in programming books is proper error handling; in production code you'd probably want to define your own error type, such as the following, and signal it instead:

```
(error 'illegal-column-value :value value :column column)
```

Then you'd want to think about where you can add restarts that might be able to recover from this condition. And, finally, in any given application you could establish condition handlers that would choose from among those restarts.

[3]If any MP3 files have malformed data in the track and year frames, **`PARSE-INTEGER`** could signal an error. One way to deal with that is to pass **`PARSE-INTEGER`** the `:junk-allowed` argument of **`T`**, which will cause it to ignore any non-numeric junk following the number and to return **`NIL`** if no number can be found in the string. Or, if you want practice at using the condition system, you could define an error and signal it from these functions when the data is malformed and also establish a few restarts to allow these functions to recover.

[4]This query will also return all the songs performed by the Dixie Chicks. If you want to limit it to songs by artists other than the Dixie Chicks, you need a more complex `:where` function. Since the `:where` argument can be any function, it's certainly possible; you could remove the Dixie Chicks' own songs with this query:

```
(let* ((dixie-chicks (matching *mp3s* :artist "Dixie Chicks"))
       (same-song (in :song (select :columns :song :from *mp3s* :where dixie-chicks)))
       (query #'(lambda (row) (and (not (funcall dixie-chicks row)) (funcall same-song row)
  (select :columns '(:artist :song) :from *mp3s* :where query))
```

This obviously isn't quite as convenient. If you were going to write an application that needed to do lots of complex queries, you might want to consider coming up with a more expressive query language.

[5]The version of **`LOOP`** implemented at M.I.T. before Common Lisp was standardized included a mechanism for extending the **`LOOP`** grammar to support iteration over new data structures. Some Common Lisp implementations that inherited their **`LOOP`** implementation from that code base may still support that facility, which would make `do-rows` and `map-rows` less necessary.