EDB Manual

The Emacs Database

by Michael Ernst

Copyright © 1991-1995 by Michael Ernst <mernst@theory.lcs.mit.edu>

This documentation describes version 1.21 of EDB, dated Jul 18 1995. The documentation was last modified on July 18, 1995.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

1 Introduction

EDB is a database program for GNU Emacs. It permits you to manipulate structured (or not-so-structured) data within Emacs and provides many of the usual database features, including:

- Flexible, customizable file layouts. Data may contain any character, including those used to delimit fields and records. Files read and written by the database may have arbitrary formats.
- Typed fields (e.g. integer, date, string); fields may also be subject to additional constraints (prime number, date before today, string that appears in some other record, etc.).
- Arbitrary data display formats for viewing records. Multiple display formats can be open on a database simultaneously, viewing the same or different records. The data display format can be automatically chosen based on the record's field values.
- Selective display of only those records of interest; others become temporarily user-invisible.
- Standard GNU Emacs editing commands, which work only within data fields and not on the surrounding text.
- Database summaries, which show in a single buffer one or more lines of information about each record.
- Sorting, with an easy-to-use a graphical interface for defining the sorting criteria; most sorting orders you would care about are easy to specify, but arbitrary ones are also permitted.
- Merging and reconciliation of databases.
- Reports generated from database information.
- Highly customizable via the underlying programming language, Emacs Lisp; many hooks and useful variables are provided to make this even easier.
- Documented by a 100-page manual.

EDB is more ambitious—and therefore more complex—than its forerunners (such as Forms Mode by Johan Vromans <jv@mh.nl>). While other packages don't provide as much functionality as EDB, they may be more appropriate for simple needs.

While EDB provides sophisticated functionality and is extensively customizable, it is not as powerful as commercial database products, and does not directly qualify as either "relational" or "object-oriented." On the other hand, EDB may well meet your needs without being buzzword-compliant. (EDB extensions to provide a subset of the features of a relational or object-oriented database can and have been written.)

1.1 Organization of this manual

This manual contains two major parts. The first part describes how to use EDB to manipulate an existing database, and the second part describes how to design a new database.

The first part—which could be called the EDB User Manual—first presents basic commands such as starting up the database, adding, deleting, and modifying records, and searching; then it describes features for the more advanced user, such as sorting, displaying record summaries, marking or ignoring certain records, and producing reports.

The second part—which could be called the EDB Database Designer Manual—describes the three forms that database information can take: when being manipulated by EDB, when stored on disk, and when displayed on the screen. Separate chapters discuss specifying each of these representations. The manual then goes on to discuss customization hooks and explains some of the lower-level implementation details that an advanced designer may need to know.

1.2 Installing EDB

EDB is available for anonymous ftp from '`theory.lcs.mit.edu:/pub/emacs/edb/`'; the package is in a single compressed tar file '`edb-`*version*`.tar.gz`', and the individual files can also be obtained from the '`code/`' subdirectory. Place the files in your Emacs load path (probably in a directory of their own). You can add a directory to your Emacs Lisp load path by putting something similar to the following in your '`.emacs`' file:

```
(setq load-path (cons (expand-file-name "~/emacs/edb") load-path))
```

Typing '`make`' in the EDB directory will byte-compile EDB and create Info files; '`make install`' does the above and also installs the Info files. You should check '`Makefile`' first to ensure that EDB gets installed where you want. It is strongly recommended that you run EDB byte-compiled, as it is sluggish when run interpreted; for more details, See Section 18.2.3 [Compiling EDB], page 94.

The texinfo documentation must be processed using release 2 of texinfo (which is available via anonymous ftp from '`prep.ai.mit.edu:/pub/gnu/`'), but the resulting Info files can be read using any Info reader. If you don't have texinfo version 2, you can get the EDB documentation pre-processed in info format from '`theory.lcs.mit.edu:/pub/emacs/edb/`', files '`edb.info`' and '`edb.info-[123456]`' (seven files in all). Similarly, that directory contains ready-to-print versions of the manual (file '`edb.dvi`' or '`edb.ps`').

EDB will do you little good without a database to manipulate; for information about creating a new database or using an existing one (EDB can handle nearly any file layout imaginable and many that aren't), see Chapter 11 [Designing a database], page 33. You may also want to use existing databases as guides, or to help familiarize yourself with EDB. A number of examples can be found in the compressed tar file '`examples.tar.Z`' or in the '`examples/`' subdirectory of '`theory.lcs.mit.edu:/pub/emacs/edb/`' (the two locations contain the same examples). You can test out function `edb-after-find-file` by performing `find-file` on '`forms-demo2-int.dat`'.

1.3 Invoking EDB

You need three files to run EDB: a data file, a format file, and an auxiliary file. The data file (usual suffix '`.dat`') contains the information that makes up the database. The format file (usual suffix '`.fmt`') specifies how the fields of a particular record appear in the data display buffer, where you may view or edit one record at a time. The auxiliary file (usual suffix '`.dba`') contains additional information about the database, such as the number of fields in each record, the layout of the data file (including what characters or strings serve as field and record separators), customizations, etc. A fourth type of file is the report format file, which is a format file used in generating reports printed on the screen, to a file, or to a printer; see Chapter 9 [Reports], page 28. Different databases may share format and auxiliary files.

For examples of data, format, and auxiliary files, a description of how to create your own, and pointers to even more information, see Section 11.1 [Creating a new database], page 33, or see the examples provided with EDB (see section "Installation" in *EDB Installation Manual*).

You can combine the format and auxiliary files (by placing the auxiliary file information at the end of the format file, in the "Local variables" section), but for simplicity we will consider the two files separately. There may be many different ways to lay out a record on the screen, so a database could have many different format files; for the time being we will concentrate on the format file which is used first, which is called the primary format file, even though it might not be the one that is used most often.

When invoking the database, you typically only need to name the data file; the names of the others are inferred from its name (see Section 15.1 [Auxiliary files], page 68) or may be mentioned explicitly by it. The `db-find-file` command starts up the database:

`M-x db-find-file`

> Read a database from *database-file*; prompts when called interactively. If the database file doesn't specify a format and the format file can't be inferred from *database-file*,

the user is prompted for it too. The user is always prompted for the format if prefix argument *prompt-for-format* is non-`nil`. If the database is already read in and *prompt-for-format* is `nil`, the existing database buffer is merely selected. When called non-interactively, argument *prompt-for-format* may be a string, the name of a format file to use.

You can use `save-buffer` (`C-x C-s`) and `write-file` (`C-x C-w`) as usual to save databases buffer to the file from which they were read or to an arbitrary file.

To autoload EDB, place the following in your '`.emacs`' file:

```
(autoload 'db-find-file "database" "EDB database package" t)
(autoload 'load-database "database" "EDB database package" t)
(autoload 'byte-compile-database "database" "EDB database package" t)
```

Now, when you start up Emacs, you will already be able to execute `db-find-file`; EDB will be loaded automatically. (You may also wish to autoload function `edb-update`; see Chapter 18 [EDB support], page 92.) See below if you wish to byte-compile the EDB sources.

Here is one way to arrange to automatically run EDB when you read a database file via the usual `find-file` command (ordinarily bound to `C-x C-f`), whether or not you choose to autoload EDB. This only works on databases which have been stored in EDB internal file layout (see Section 13.1 [Internal file layout], page 43). The reason `edb-after-find-file` isn't built into EDB is that loading the definition would require loading EDB, which is undesirable when only ordinary editing will be done.

```
(setq find-file-hooks (cons 'edb-after-find-file find-file-hooks))
(defun edb-after-find-file ()
  "If this is a database file in EDB internal file layout, run EDB.
To be placed in `find-file-hooks'."
  ;; When this is called, we are at the beginning of the buffer.
  (if (looking-at ";; Database file written by EDB")
      (progn
        (require 'database)
        (db-this-buffer)
        ;; db-this-buffer kills the current buffer; and an error results
        ;; when Emacs tries to switch back to it.  find-file-noselect
        ;; uses the buf variable to hold the new buffer.
        (setq buf (buffer-name (current-buffer))))))
```

1.4 Example EDB session

This section describes some of the most frequently used EDB commands. All of the commands used here are more fully documented elsewhere in the manual. For a very brief introduction to database mode, its submodes (Database View mode and Database Edit mode), the data display buffer, EDB concepts, and more, see Chapter 2 [Database mode], page 7.

`M-x db-find-file RET forms-demo2.dat RET`
: Load the database. After calling `db-find-file`, you may be asked for the name of the format file if EDB can't infer it from the name or contents of the data file. See Section 1.3 [Invoking EDB], page 3. When the database finishes loading, the data display buffer is visible and EDB is in Database View mode (see Chapter 3 [Database View mode], page 9).

`n`
`p`
: Go to the next or previous record (see Section 3.1 [Moving around in the database], page 9). The data display buffer is in Database View mode, which is read-only and does not permit editing (see Chapter 3 [Database View mode], page 9).

`TAB`
`C-n`
: Go to the first field and switch to Database Edit mode. See Section 3.2 [Changing to Database Edit mode], page 10, and Chapter 4 [Database Edit mode], page 13. Once you are on a field, printing characters insert themselves and the other usual editing commands work as well. `TAB` moves to the next field, and `C-n` moves to a field on the next line (or to the next line of this field, if it spans multiple lines). See Section 4.4 [Moving from field to field], page 14, and Section 4.5 [Movement within a field], page 15.

`M-TAB`
`C-p`
: Like `TAB` and `C-n`, but move backward by fields or lines.

`M-s`
: Search for a value in the current field. See Chapter 5 [Searching], page 16.

`C-c C-c`
: Return to Database View mode.

`h`
: See a summary of all the records of the database; `h` stands for "headers". See Chapter 7 [Database Summary mode], page 23. You can move around in the summary buffer using ordinary movement commands, and the record under point will be displayed in the data display buffer. Use `v` or `e` to return to the data display buffer; `v` puts you in Database View mode and `e` puts you in Database Edit mode.

`M-x db-sort RET`
: Invoke the database sort interface, which permits easy specification of how records should be sorted, and then perform the sort. Type `C-h m` or `?` for help while in the database sort interface (or anywhere in EDB). See Chapter 6 [Sorting], page 19.

`C-x C-s`
: Save to disk any changes you have made to the database.

`C-h m` Display a list of commands. This command works in all Emacs modes (and in particular, in all EDB modes).

`q` Quit EDB. You are offered the chance to save the changes you have made, then the database buffer is buried.

EDB provides many more commands than these; see this manual's table of contents or index (see [Concept Index], page 101) to find the topic that interests you.

1.5 Terminology

A database is a collection of records, each of which is comprised of fields. A record's fields are usually all related to some central object or concept; for instance, they might describe various information about a particular person such as name, address, and phone number. All records of a database have the same structure (they contain the same fields), though typically different records have different information in the fields.

EDB permits database records to be viewed, edited, and manipulated in a structured way.

2 Database mode

A single database record (typically the "current record") is viewed in a data display buffer.[1] The layout and formatting of the data display buffer—where and how the fields of the current record are shown, and what fixed explanatory text surrounds them—is specified by a data display format. Only the database fields can be edited; the explanatory text is fixed. Creating a new data display format is described in Chapter 10 [Specifying the display format], page 29. Creating a new data display buffer (with the same or a different data display format) is described in Section 10.3 [Making additional data display buffers], page 31. Viewing summary information about all database records at once is described in Chapter 7 [Database Summary mode], page 23.

Database mode has two basic submodes, Database View mode (see Chapter 3 [Database View mode], page 9) and Database Edit mode (see Chapter 4 [Database Edit mode], page 13). These modes are used, respectively, when examining or manipulating records and when changing information in a particular record. Keystrokes have different meanings in these two modes. In Database View mode no editing may be done, and many printable characters are redefined to make manipulation of the database easy (for instance, `n` moves to the next record). In Database Edit mode a field of the current record is being edited; most printable keys insert themselves, and other editing and movement commands work in the ordinary way. In the data display buffer, where database records are ordinarily viewed and edited, one of these two modes is always in effect. (You may be tempted to directly edit a raw database file in its on-disk layout. Do so at your own risk, and never change a database buffer out of Database mode.)

The mode line indicates which mode you are in. It looks something like:

```
-***-Database: machine-dbase        (Edit Abbrev 42/431)---All------------
```

The mode line consists of three modification indicators, the word '`Database:`' (which reminds you that you are in database mode), the name of the database file being manipulated (in this case, '`machine-dbase`'), minor mode information within parentheses, and the usual percentage-of-screen-visible indicator. The minor mode information consists of the database submode (such as View, Edit, or Summary), any other minor modes which are turned on (such as Abbrev mode), the number of the current record, and the total number of records in the database.

Ordinarily the Emacs mode line contains only one modification indicator consisting of two dashes (not modified), asterisks (modified), or percent signs (read-only). The EDB mode line

[1] The data display buffer was previously called the format buffer; this is why all of the variables and functions relating to it start with the `dbf-` prefix.

contains three modification indicators, one each for the database, the displayed record, and the current field. The field indicator is '`*`' if the field under point has been modified, '`-`' if it has not, and '`%`' if it is read-only or if no field is under point (for instance, if the data display buffer is in Database View mode rather than Database Edit mode).

The database is modified only when a changed record is written into it; changes to the displayed record (also called the current record) do not immediately affect the database proper. This permits such modifications to be conveniently undone. (See Section 3.3 [Undoing all changes to a record], page 10, and Section 3.4 [Making changes permanent], page 10.) Thus, the displayed record may be modified without the database being modified, since the database is considered modified only when the current record has been processed and the resulting value placed in the database. A similar situation exists for the current field and the displayed record. (See Section 4.2 [Undoing changes to a field], page 13.)

Do not attempt to directly change the major mode of a database buffer. If a database buffer is placed in another mode, the database functions will cease working (they refuse to operate on non-database buffers, since the consequences of such action could be severe); for instance, you may be unable to save any of your work due to errors raised in the execution of `save-some-buffers`. Furthermore, EDB makes assumptions about where point is located in Database View and Database Edit modes; violating these can cause changes to the current record to be lost.

3 Database View mode

The data display buffer is in Database View mode whenever field information is not being edited. Most commands to move from record to record and to manipulate records (sorting, printing reports, showing summaries, etc.) are performed in Database View mode.

Basic operations are described here; more complicated ones, such as searching (see Chapter 5 [Searching], page 16), are given sections of their own.

3.1 Moving around in the database

`n` Go to the *arg*th next record (`db-next-record`). In that record, go to the current field, if any.

`p` Go to the *arg*th previous record (`db-previous-record`). In that record, go to the current field, if any.

`<`
`M-<` Show the database's first record (`db-first-record`). With optional prefix argument, ignores hiding.

`>`
`M->` Show the database's last record (`db-last-record`). With optional prefix argument, ignores hiding.

`j` Show the database's *arg*th record (`db-jump-to-record`). Hiding is ignored unless optional argument *respect-hiding* is specified.

There are two special hybrid commands that show more of the current record if there's more to see and otherwise show the next (or previous) record.

`SPC` Go to the *arg*th next screenful of this display, or to the *arg*th next record, if this is the last screenful of this display (`db-next-screen-or-record`). If point is in the summary buffer and the data display buffer is not visible, then move to the next record.

`DEL` Go to the *arg*th previous screenful of this display, or to the *arg*th previous record, if this is the first screenful of this display (`db-previous-screen-or-record`). If point is in the summary buffer and the data display buffer is not visible, then move to the previous record.

3.2 Changing to Database Edit mode

When in Database View mode, you cannot edit the fields of the record being displayed. In order to do so, change to Database Edit mode and move to the field you wish to edit. You can click the mouse on the field you wish to edit, or move to the first or last field (and from there to the desired field) via the following keystrokes:

`TAB`
`C-n`
`e` Move to first field (`db-first-field`).

`C-p`
`M-TAB` Move to last field (`db-last-field`).

3.3 Undoing all changes to a record

`C-x u` Set the record to be the same as the corresponding one in the database (`db-revert-record`). In other words, undo any changes made since entering this record.

`C-x r` Replace the database with the data on disk (`db-revert-database`). This undoes all changes since the database was last saved.

You can also undo changes to the current field; see Section 4.2 [Undoing changes to a field], page 13.

3.4 Making changes permanent

You edit a copy of a database record; the database itself is not changed until you commit the changes. This occurs automatically whenever any command causes a different record to be displayed, when the database is saved, when a report is generated, and so forth. It does *not* occur when you switch from Database Edit mode to Database View mode, though the field modification flag (in the mode line) will become a percent sign and the record modification flag, an asterisk. When the record is committed, the record modification flag becomes a dash and the database modification flag becomes an asterisk.

You can manually install the current record, as modified, into the database. The following two functions are identical:

`RET` Install the current record in the database; make any changes permanent (`db-accept-record`).

`M-x db-commit-record`
Install the current record in the database; make any changes permanent.

Committing a record makes the changes permanent only insofar as they become part of the in-memory representation of the database. The on-disk version is not affected unless you overwrite it by using `save-buffer` or `write-file`, or otherwise indicate that the database should be written to disk (say, by responding affirmatively to a question about saving the database).

3.5 Adding and removing records

`a`
`i` Add a new record to the database immediately before the current record (`db-add-record`).

`c` Insert a copy of the current record in the database immediately after it (`db-copy-record`). The second of the two records is made the current record. With a prefix argument, inserts that many copies.

`o` Copy (output) the current record to *database* (`db-output-record-to-db`). *database* must be read in and compatible with the current database.

`d`
`k` Remove the current record from the database (`db-delete-record`). With a prefix argument, doesn't verify.

By default, deleting a record marks the database as modified. Set the following variable to change this behavior.

`db-delete-record-modifies-database-p`
Non-`nil` if deleting a record should mark the database as modified.

3.6 Exiting database mode

`q` Quit editing the database for now; bury its buffers (`db-quit`).

You can also kill a database buffer in the usual way (for instance, by using `kill-buffer`); this causes `db-kill-buffers` to be called. If the database's last buffer is killed this way, the database itself is also killed. No offer is made to save changes; call `db-quit` in order to do that.

`db-kill-buffers`

Kill this buffer, and the associated summary or data display buffer, if any. If its last data display buffer is killed, the database is killed too. Does not offer to save changes to the database or to this record; use `db-exit` with optional argument to do so.

4 Database Edit mode

The data display buffer is in Database Edit mode whenever a field of the current record is being edited. All printing characters insert themselves, and the usual editing commands can be performed on the field contents. It is also possible to move from field to field and from record to record, and to perform other operations on the database, though the commands available in Database View mode are richer.

In Database Edit mode, point is always in the field currently being edited. The database is not modified as soon as changes are made in Database Edit mode. Rather, a copy of the record in question is displayed and edited, and only when you move to a new record, initiates some other global action (not specific to the edited record), or explicitly commits the changes (see Section 3.4 [Making changes permanent], page 10). This permits easier undoing of incorrect modifications.

In order to perform most record-level operations, you switch to Database View mode, and then perform them there. Several commonly used commands, however, such as searching and moving from record to record, are accessible directly from Database Edit mode.

Commands that move from field to field check the validity of the current field before moving off it; commands that move from record to record do this as well, then make any changes in the current record permanent (though the database file on disk is not changed).

Basic operations are described here; more complicated ones are given sections of their own.

4.1 Exiting Database Edit mode

`C-c C-c` Switch to Database View mode (`db-view-mode`). With an argument, toggle between Database View and Database Edit modes.

4.2 Undoing changes to a field

You can undo changes to the current field via Emacs' usual undo facility; use `C-x u` or `C-_` to undo changes made since entering the current field.

You can also revert the current field to its original value; this is useful if you made a change, moved off the field, and then moved back onto it.

`C-x U` Undo any changes made since entering this field (`db-revert-field`). Replace the onscreen text in this field with that of the underlying record.

A similar effect can be had by invoking `C-x u` multiple times.

From Database View mode, you can simultaneously revert every modified field of a record to its original value; see Section 3.3 [Undoing all changes to a record], page 10.

4.3 Moving from record to record

These commands make any changes to the current record permanent.

`M-n` Go to the *arg*th next record (`db-next-record`). In that record, go to the current field, if any.

`M-p` Go to the *arg*th previous record (`db-previous-record`). In that record, go to the current field, if any.

4.4 Moving from field to field

`TAB` Move to *arg*th next reachable field, wrapping if necessary (`db-next-field`). When called interactively, *arg* defaults to 1.

`M-TAB` Move to *arg*th previous reachable field, wrapping if necessary (`db-previous-field`). When called interactively, *arg* defaults to 1.

`M-<` Move to first field (`db-first-field`).

`M->` Move to last field (`db-last-field`).

Also see the keystrokes `C-n` and `C-p`, described below.

4.5 Movement within a field

All Emacs cursor motion commands retain their standard meanings, except that they do not move outside the field.

The line-movement commands have slightly changed meanings: if the motion would take the cursor out of the current field, then they move to the next field.

`C-n` Move to *arg*th next line (`db-next-line-or-field`). If that would move out of the current field, move to the closest field to that, but not the current one, wrapping if necessary.

`C-p` Move to *arg*th previous line (`db-previous-line-or-field`). If that would move out of the current field, move to the closest field to that, but not the current one, wrapping if necessary.

4.6 Editing a field

Many Emacs editing commands retain their standard meanings; for instance, printing characters insert themselves and deletion commands work as usual, except that they do not make changes outside the field; among these are `C-d` (`delete-char`), `DEL` (`backward-delete-char`), `M-d` (`kill-word`), `M-DEL` (`backward-kill-word`), and `C-k` (`kill-line`).

4.7 Getting help

You can get some information about the current field, such as what type of value it expects or what its contents signify, by using the following command.

`M-?` Display help for current field using the recordfieldspec help-info field (`db-field-help`). If this is a string, display it. If it is a form, eval it and display the result.

5 Searching

A useful and commonly used database operation is searching for records that meet some criteria: for instance, finding a particular record or indicating that following operations should only apply to records that correspond to an address in greater Boston. EDB provides several functions to support such operations.

To perform a search pertaining to the contents of only one field, move to that field and use the following command:

`M-s` Search for occurrences of *pattern* in the current field of any record (`db-search-field`). Finds the first match after the current record; wraps around automatically. With prefix argument, marks all matches in addition to going to the first one. If hiding is in effect, hidden records are ignored.

The same keystroke in Database View mode permits specification of patterns which depend upon the contents of several fields:

`s`
`M-s`
`M-S`

`db-search` is not yet implemented; use `db-search-field` instead (`db-search`). In a future version of EDB, `db-search` will permit searching on all fields of a record simultaneously.

For a description of marking, see Chapter 8 [Marking and hiding], page 25.

5.1 Search patterns

Search patterns can be as simple as a datum to match exactly or as complicated as the conjunction, disjunction, and negation of tests to be performed on field contents.

5.1.1 Basic patterns

A basic search pattern has the same form as the data that is kept in the field; for instance, to search in a string field for a particular string, use that string; to search for the date March 14, 1967, use '`3/14/67`' or '`14 March 1967`' or any other accepted date format. A basic pattern is treated somewhat more richly than a literal, however. In a string field, typing a string results in a match for any element which contains it as a substring; typing '`ail`', without the quotes of course, matches both "ailment" and "fail". In a date field, '`3/67`' matches any date in March of 67, not just those March 67 dates which specifically exclude a day of the month.

5.1.2 Comparisons

A search pattern may also be a comparison prefix (`<`, `>`, or `=`) plus a datum which is treated exactly like any other element of that field. In a string field, '`<ail`' matches all elements lexicographically less than "ail"; '`=ail`' matches only fields containing exactly "ail", but not "ailment" or "fail". Warning: '`= ail`' looks for an entry containing " ail", that is, four-character sequence starting with a space.

In a date field, searches work slightly differently; for instance, '`>3/14`' matches dates after March 14 in any year, and '`=3/67`' matches only dates in March 67 whose day of month is not specified. For more information about the interpretation of patterns, see the documentation for the particular types.

5.1.3 Logical connectives

More complicated patterns can be built up out of simpler ones via the logical connectives AND, OR, and NOT. These work in the obvious way. One pattern which finds any date between the ides of March and Christmas, inclusive, is '`> 3/14 AND < 12/26`'; two patterns which find dates except March 14 are '`NOT 3/14`' and '`< 3/14 OR > 3/14`'. To find strings that either contain the substring "ail" or start with a, b, or c, use '`ail OR < d`'.

The precedence of these connectives is: NOT, which is most tightly bound to its test, then OR, then AND. There are no provisions for grouping or otherwise overriding this ordering. Connectives (and REGEXP, described below) consume all surrounding spaces and tabs.

5.1.4 Other pattern operations

One other pattern operation of interest is the regexp operator for string fields. This is invoked by either using REGEXP with surrounding spaces, or / without a trailing space. For instance, '`/^[ace]`' matches any string field starting with a, c, or e; '`/.`' matches any nonblank string field; and '`NOT REGEXP a.*b.*c`' matches any string which does not contain the letters a, b, and c in order. This last example shows that EDB's search commands are more powerful than general regular expression searching.

6 Sorting

Sorting a database reorders its records without changing the contents of any particular record. For instance, it might be convenient to arrange an address database sometimes alphabetically by last name, sometimes by ZIP code, and sometimes by some other criteria.

`M-x db-sort`
: Sort the database. With a prefix argument, don't confirm the sort order.

Ordinarily, calling this function invokes the sort interface, a graphical tool for specifying which fields should be treated as sort keys, and in what order. In a database whose records have fields '`firstname`', '`lastname`', '`address`', '`city`', and '`zip`', and in which records should first be sorted on '`lastname`' in increasing order, then on '`firstname`' in *decreasing* order, ignoring the other fields entirely for the purposes of the sort, the sort interface display would look like

```
==== Significant fields:
  lastname             increasing
  firstname            decreasing
==== Nonsignificant fields:
  address              increasing
  city                 increasing
  zip                  increasing
==== Hidden records to end:  No
```

In addition to which fields should be sorted by, the sort interface permits specification of how hidden records should be treated.

`t`
: Toggle the boolean value of `dbsi-hidden-to-end-p` (`dbsi-toggle-hidden-to-end`). This controls whether hidden records should all be placed at the end of the sorted order or should be sorted according to the same criteria as non-hidden records.

To change the relative order of fields, and whether they're significant or not, use the following commands.

`C-k` Kill field on current line, placing it in the sort interface kill stack (`dbsi-kill-line`).

`C-y` Yank most recently killed (lifo ordering) field, inserting it before point (`dbsi-yank-line`). This removes the field from the sort interface kill stack.

To specify how a particular field should be ordered, use the following commands.

`i` Specify that the field at point should use an increasing ordering (`dbsi-increasing`).

`d` Specify that the field at point should use a decreasing ordering (`dbsi-decreasing`).

`o` Specify an ordering function for the field at point (`dbsi-ordering-function`). An "ordering function" returns -1, 0, or 1 depending on whether its first argument is less than, equivalent to, or greater than its second argument.

`s` Specify a sorting function for the field at point (`dbsi-sorting-function`). A "sorting function" returns `t` if its first argument is less than its second argument and `nil` otherwise.

Each database has a default sort order in its `field-priorities` slot (see Section 16.1 [The database structure], page 79) which is used when setting up the sort interface (and is used for sorting when no other ordering information is specified). When you exit the sort interface, that slot can be set to the ordering depicted on the screen (see below).

Each data display buffer also has a default sort order: the database's `field-priorities` slot is ignored if the buffer-local variable `dbf-field-priorities` is `nil`. You may set or unset this variable using the sort interface; when it is unset, the default sort order is taken from the database's `field-priorities` slot. Setting the database's sort order automatically clears the `dbf-field-priorities` variable.

`dbf-field-priorities`
: The list of field priorities for this database in this data display buffer. If non-`nil`, overrides the database's field-priorities slot.

`dbf-hidden-to-end-p`
: The default, local to this data display buffer, for the hidden-to-end-p database slot. Only used if dbf-field-priorities is non-`nil`.

The following commands are used to exit the sort interface; most of them also cause the database to be sorted. Some of them set the default ordering for the database or the data display buffer.

`RET`

`C-c C-c` Use the current ordering to sort, and make it the default for future sorts of this database (`dbsi-use-ordering-make-database-default`). The user is warned if there are killed, non-yanked fields.

`A`

`U` Use the current ordering to sort, and make it the default for future sorts in this data display buffer only (`dbsi-use-ordering-make-buffer-default`). The user is warned if there are killed, non-yanked fields.

`a`
`u` Use the current ordering for this sort only (`dbsi-use-ordering`).

`!` Sort according to only the field at point (`dbsi-this-field-only`). All editing of other fields is ignored.

`q` Abort the sort and exit the sort interface (`dbsi-quit`).

`c` Clear the default sort order for this buffer and exit the sort interface without sorting (`dbsi-quit-clear-buffer-default`). In the future, the default sort order will come from the database.

The sort interface returns a field priorities list to be used when sorting; when the sort interface is entered, either the value of the `dbf-field-priorities` variable, or the database's `field-priorities` slot, is being used. For information about the format of the field priorities list, see Section 16.1 [The database structure], page 79.

Sorting does not ordinarily mark the database as modified, because not the data itself, but only the way it is arranged, has been changed. If you set `db-sort-modifies-p` to non-`nil`, then whenever a database is sorted (even if the resulting order is the same as the original one), the database is marked as modified.

`db-sort-modifies-p`
If non-`nil`, then sorting a database marks it as modified too.

6.1 Sorting and ordering functions

In order to specify the relative order of two field values (for the same field, but from different records), the database designer provides a sorting function, an ordering function, or both. If only one is provided, the other is automatically generated from it. In any event, only one of them is used for a given field on any particular sort.

A sorting function takes two field values as arguments and returns `t` if its first argument is less than its second argument (that is, the first argument appears previous to the second in the sorted order). The sorting function returns `nil` if the arguments are equal or if the first argument is greater than the second (appears later in the sorted order).

An ordering function, on the other hand, returns complete information about the relative order of its two arguments: it returns -1, 0, or 1 depending on whether its first argument is less than, equivalent to, or greater than its second argument.

Use of an ordering function can result in fewer comparisons in some cases, because it returns more information. This is worthwhile if a significant amount of processing is required before the comparison is done. For example, suppose that two addresses are being carefully checked for equality and some of the steps leading up to that are expansion of abbreviations, standardization of spelling, capitalization, and spacing, etc.; then it is better to return the exact relative ordering than to possibly require another time-consuming operation to determine it.

If it is possible to canonicalize the values beforehand, that may be even more efficient, but that is not always possible; consider the case of a small (but tedious to extract) part of the information in each field being compared.

7 Database Summary mode

A summary is a listing containing abbreviated information about every record; it permits many records to be viewed at once.

This is available in Database View mode via the following command:

`D`
`H`
`h`
: Display a summary (or directory) of all database records according to the variable `dbf-summary-function`, which is set by `dbf-set-summary-format` (`db-summary`). The summary appears in a separate buffer. When called from the summary buffer, this updates the summary.

When a summary is created, the summary display format appears in the summary buffer once for each record, with appropriate values substituted for its display specifications. Hidden records are included in the summary only if the data display buffer variable `dbf-summary-show-hidden-records-p` is non-`nil`.

`dbf-summary-show-hidden-records-p`
: Nil if hidden records should be hidden from the summary, `t` otherwise.

The entire format is indented by two characters; the first and second columns contain '`+`' and '`[`', respectively, if the record is marked or hidden. For information about marking and hiding records, see Chapter 8 [Marking and hiding], page 25.

The summary buffer is not updated whenever a record value changes; in the interest of efficiency, it remains as is until the next `db-summary` command is issued, at which time the summaries are redisplayed after all, some, or none of them have been recomputed. For the same reason, when a single mark or hide bit changes, the summary is updated; when many change, it is usually not.

When point is in the summary buffer, the associated data display buffer (nearly) always displays the record under point. (Some Emacs commands can move point without EDB noticing.) Movement in the summary buffer is by any of the ordinary Emacs commands, including searching. Most Database View mode commands also work in the summary buffer.

The summary display format defaults to the first non-literal line in the database display format—that is, the first line which contains a display specification.

Summary display formats can display information any way that an ordinary display format can, including showing more than one field of the record in question or spanning several lines. The only restriction is that the summary display format must cover a specific number of lines: each display specification must have its `min-height` and `max-height` slots set to equal values. For more information about display formats, See Chapter 14 [How information is displayed], page 58.

The following command may appear in a format file or an auxiliary file, or it can be invoked directly. When it appears as an Emacs Lisp form, remember the special meaning of the backslash character and double it where necessary.

`M-x dbf-set-summary-format`

Specify the format used in the Database Summary buffer. Argument *summary-format* is a string containing display specifications. Call this in the data display buffer, or in a format file or auxiliary file.

8 Marking and hiding

The marking facility permits operations to be performed on only certain records of a database. For instance, to create a report which describes only some of the database records, you would first mark the records of interest. Then you would call `db-report` with a prefix argument (do so by pressing `C-u` first), or would make optional second argument *markedp* non-`nil`. See the documentation of the individual operations to see whether they support operation on only the marked records.

The hiding facility also restricts attention to a subset of the current database. It is similar to Outline mode in that it makes some records temporarily invisible. Hiding is useful when you wish to concentrate on a subset of the database without being distracted by other records that may be present. By default hidden records are skipped by the record-motion commands, excluded from searches and reports, and ignored by most other operations.

Mark and hide bits are associated only with the in-memory version of a database; they are not saved when you save a database, and are always unset (boolean false) when a database is read in. (When a database is written to disk, although the disk version of the data does not contain the mark and hide information, that information is not lost from the working copy of the database.)

This behavior is a feature, not a bug. Marking and hiding are intended to help you temporarily group database records for operations upon them; if there is information that cannot be recreated from a record's fields, then you should consider adding another field for that information. On the other hand, mark or hide criteria may be complicated. If such a pattern is used often, then you may wish to write a function to set the bits appropriately, which function could be bound to a keystroke or automatically executed when the database is read in.

8.1 Setting the mark and hide bits

Every record may be thought of as having a pair of bits or boolean values indicating whether it is marked and whether it is hidden. The most straightforward way to set these bits is to use an operation to mark, unmark, hide, or unhide a particular record; these are bound to keystrokes in Database View mode.

`m` Toggle whether the current record is marked (`db-mark-record`). With a nonzero prefix argument, set it to be marked. With a zero prefix argument, set it to be unmarked.

`O` Change whether the current record is hidden (`db-hide-record`). With a nonzero prefix argument, set it to be hidden. With a zero prefix argument, set it to be unhidden.

The searching commands, when called with a prefix argument, mark each matching record; Chapter 5 [Searching], page 16.

Once all records of interest have been marked, through one or more marking and/or searching commands, unmarked records can be hidden from consideration. This is useful if you want to work on only a small number of records, or if specifying the records of interest is easier than specifying those not of interest: instead of hiding all the uninteresting records, simply mark the interesting ones, then use the following command to cause the unmarked ones to become hidden.

`M-x db-hide-unmarked-records`
Hide all unmarked records. Also clears all mark bits and sets `dbc-hide-p`.

The converse operation transfers information from the hide bits to the mark bits.

`M-x db-mark-unhidden-records`
Mark all unhidden records. Also clears all hide bits.

It is also possible to clear all the mark or hide bits.

`M-x db-unmark-all`
Clear the mark bit of every record.

`M-x db-unhide-all`
Clear the hide bit of every record.

8.2 Movement among marked and hidden records

Ordinarily, record movement commands (those which move from one record to another) ignore hidden records, so that you never land on an hidden record. Marked records, on the other hand, are not treated specially by the record movement commands. The following Database View mode keystrokes permit you to move to hidden records or to move directly to marked records.

`M-n` Go to the *arg*th next record, ignoring omissions (`db-next-record-ignore-hiding`). That is, all records, even those which are hidden, are counted.

`M-p` Go to the *arg*th previous record, ignoring omissions (`db-previous-record-ignore-hiding`). That is, all records, even those which are hidden, are counted.

`M-C-n` Go to the *arg*th next marked record (`db-next-marked-record`). Hidden records are treated according to db-hide-p.

`M-C-p` Go to the *arg*th previous marked record (`db-previous-marked-record`). Hidden records are treated according to db-hide-p.

8.3 Details of hiding

The "hidden" bit of each record has no effect unless the following variable is set:

`dbc-hide-p`
: Non-`nil` if hiding is in effect, `nil` otherwise. Use function `dbc-set-hide-p`, which works in either a data display buffer or a summary buffer and sets the variable's value in both, instead of setting this directly. Setting this to `nil` is cheaper than changing the hide function to the empty one, since no hide bits are recomputed. This variable is automatically set by the hiding functions.

When `dbc-hide-p` is `nil`, the values of records' hide bits are remembered, and they may still be set and unset, but they have no effect on any operations until `dbc-hide-p` is once again set to non-`nil`. When `dbc-hide-p` is non-`nil`, '`Hide`' appears in the mode line of the database buffer.

The following operations work in either Database View mode or Database Summary mode.

`M-o` Change whether hiding is in effect (`db-hiding-toggle`). With a nonzero prefix argument, turn hiding on. With a zero prefix argument, turn hiding off.

This does not change the current hide-function, and a hide bit is always computed for each record, but hide bits have no effect on any operations if hiding is not in effect.

`M-C-o` Toggle whether hidden records are shown in the summary (`db-toggle-show-hidden-records`). With a nonzero prefix argument, show hidden records in the summary. With a zero prefix argument, don't show hidden records in the summary.

`M-O` Set the criteria for automatically determining whether to hide a record (`db-hiding-set`). This isn't implemented yet.

9 Reports

Reports can be generated from a database by using the following command in view mode:

`r` Create a report according to *report-filename* (`db-report`). Prefix argument *markedp*, if non-`nil`, means report on only marked records. If hiding is in effect, hidden records are not reported upon. When called interactively, prompts for *report-filename*.

The way a report looks is specified in precisely the same as are display formats and summary formats (see Chapter 14 [How information is displayed], page 58). This information must be placed in a file; you cannot type it directly when creating a report. This restriction makes errors in the report format easier to correct.

The report is placed in the '`*Database Report*`' buffer, which is in Text mode. The information may then be edited, saved to disk, or otherwise manipulated. The buffer is in Text mode and is not yet a file; you must save it to make it a file.

To create a report which mentions only marked records (see Chapter 8 [Marking and hiding], page 25), supply a prefix argument to the report command by typing `C-u` first.

9.1 Bugs in report generation

There are currently a few unfortunate bugs in EDB's handling of reports. The most noticeable of these are errors in the handling of format information: primarily, indentation is not respected. Bug fixes for these problems are welcome.

10 Specifying the display format

Different layouts and on-screen arrangements of the values stored in database records are appropriate when you are concentrating on different aspects of the data. Sometimes you prefer to see just a few of the fields; at other times you may want to see the records in full detail. It may also be appropriate for the display format of a record to depend on the record's field values. This section describes how to choose a different display format for the record being displayed, either manually or automatically.

10.1 Changing display formats

EDB permits the creation and use of a variety of display formats with a single database; you can also conveniently change the way that a particular record is displayed by using `db-change-format` and specifying the filename of the new display format, or a nickname for the format that has been specified by the user or the database designer. Choosing a different format does not create a new data display buffer; it changes the way that records are displayed in the current one.

`M-x db-change-format`

Select and use an alternate display format to view the database. If neither *format-name* nor *filename* is specified (as is the case when this is called interactively), the user is prompted for them. In Emacs Lisp code, if `dbf-format-name-spec-alist` has been been set, usually only one of the arguments is specified. If both are specified, then *format-name* becomes a name for the format *filename* specifies; if *format-name* is already associated with a different format file, an error is signalled.

If the current format is unnamed, the user is prompted for a name to give it, so that it can be conveniently restored if need be. This behavior is suppressed, and the record is not displayed, if the function is not being called interactively.

The data display buffer is left in Database View mode.

Selecting the current format does not cause any work to be done.

Some databases automatically set the format of the record being displayed, usually by setting `dbf-before-display-record-function` to a function that overrides the format in effect when a record is about to be displayed. This may cause this function to appear not to be doing any work. In actuality the format is being set, then reset.

`dbf-format-name-spec-alist`

Association list of format names and format specifiers. Each format name is an arbitrary string. A format specifier is a filename or format file specifier, which is a list

of values for format variables. The user sets the format specifier to a filename, and after that format file has been read, EDB replaces the filename with a list of values for format variables, so that the file need not be read again.

It is convenient for a database designer to set this, pre-assigning format names to files so that the user only needs to remember the format names, not the filenames.

`dbf-format-name`
: This buffer-local string names the format currently in use. This should not be set by the user.

`dbf-format-file`
: The format file from which the current format was built.

These values can be set in the auxiliary or format files so that you can choose a format name (with completion) instead of having to remember a filename. If the selected format's specifier is a filename, then after the file is read in, the format-spec is modified by replacing the filename with information about the format such as the displayspecs, the invariant text between them, and so forth. Subsequent selections of that format do not cause disk accesses. For an example of the use of `db-change-format`, see Section 15.2.3 [Record display hooks], page 71.

While it is not currently possible to selectively hide certain fields from a data display buffer, judicious use of alternate formats can result in nearly the same effect.

10.2 Execution of format file eval expressions

Often the "Local Variables" section of a format file contains code that should only be executed once, or should only be executed before the database is read in, because of either efficiency or correctness constraints. Because of this, the "Local Variables" section of a format file is executed only when it is read in from disk (which is usually only once). In order to cause an expression to be evaluated every time that a particular display format is selected, use the `dbf-always` macro:

`dbf-always`
: Execute *body*, and place its forms in `dbf-always-forms`. They will be executed each time that this format replaces another.

`dbf-always-forms`
: Forms executed every time that the format is selected. These forms are only executed when a different format is replaced, not every time that a record is displayed (or even every time that `db-change-format` is called). See also `dbf-before-display-record-function`.

Of course, it is often valuable to overwrite a value when the display format changes; this is the purpose of `dbf-always`. It is always safe to set variables whose name begins with `dbf` in such forms, though changes to some such values—none of which the user should be changing anyway—will not take effect when a display format is being returned to (though they will work when it is first chosen). This will affect you only in that a call to `dbf-set-summary-format` will have an effect only the first time that a format file is read in, not every time that it replaces another, even if it is enclosed in a `dbf-always` form.

The forms in `dbf-always-forms` are not executed every time that a record is displayed, or even every time that `db-change-format` is called, but only when a format replaces another one (that is, `db-change-format` is called and its first argument is not equal to `dbf-format-name`).

Here is an example of a common problem with an expression which causes an error if evaluated every time that the format is selected. The primary format file (the one that is used when the database is read in) is permitted to set the `fieldnames` slot of the database structure to a list; other parts of the database initialization code propagate that information into other slots of the database structure and change the list into a vector, which is its proper representation. If you switch to another display format and back to the primary one, and the `database-set-fieldnames` expression was evaluated, then the next attempt to access the `fieldnames` slot of the database as a vector would cause an error. In this case, the proper solution is to use `database-set-fieldnames-to-list` (see Section 16.1 [The database structure], page 79) instead, but such functions are not provided for every slot that it would be dangerous to set. If several format files all set a value which is dangerous to change, then another possibility is to check the value before setting it: if it is already set, then don't do anything. Another possibility is to move all assignments to database slot values from the format file to the auxiliary file.

10.3 Making additional data display buffers

In addition to changing the display format of an existing data display buffer, it is sometimes useful to have two different data display buffers both examining the same database, either so that two different records can be viewed or edited simultaneously or so that two different formats can be used at the same time—or both.

Use the following function to create a second (or additional) data display buffer for the current database.

`M-x db-additional-data-display-buffer`

Create another data display buffer in which to view this database.

If you edit the same record in more than one data display buffer, only the last one committed (by calling `db-commit-record`, moving to another record, saving the database, etc.) has an effect. (Simply switching from Database Edit mode to Database View mode does *not* commit the changes; failing to commit changes makes it appear that changes in one data display buffer are not being communicated to the other ones associated with the same database. For more on committing, see Section 3.4 [Making changes permanent], page 10.) It is perfectly safe, however, to edit different records of the same database in different data display buffers, or to perform any other database manipulations.

To learn about formatting directives and specification of display format files, see Chapter 14 [How information is displayed], page 58.

11 Designing a database

To set up a database for editing, the *database designer* must specify pertinent information about the database, including

- the number of fields per record and the type of each one (see Chapter 12 [Record field types], page 37);
- the layout of the file containing the database (see Chapter 13 [Database file layout], page 43);
- how each record should be displayed on the screen when it is being viewed or edited (see Chapter 14 [How information is displayed], page 58); and
- special actions to be performed, such as updating the '`last-edit-date`' field of a record whenever any other field is modified (see Chapter 15 [Customization], page 68).

Detailed information about the database's internal representation is supplied in Chapter 16 [Database representation], page 79.

11.1 Creating a new database

This section tells you how to quickly create a trivial database; please don't be satisfied with this, but follow the references to learn how to access EDB's more sophisticated features. Another good way to learn about EDB is to look at the example databases provided with EDB (see section "Installation" in *EDB Installation Manual*).

First, you must decide how many fields the database shall contain, and what their names are. The names are Lisp symbols, and they are primarily used internally—most users never know how EDB refers to the fields. Let us make a name database with three string fields called '`first`', '`middle`', and '`last`' and one integer field called '`age`'.

As mentioned in Section 1.3 [Invoking EDB], page 3, you need three files to run EDB: a data file, a format file, and an auxiliary file. The auxiliary file is optional; its functionality can be placed in the format file.

The data file may have any one of a number of layouts. Perhaps the simplest is tab-delimited text, which is also the default: fields are separated by tab characters and records are separated by newlines. EDB doesn't deal well with empty databases (because it always tries to display a record in the data display buffer), so create a file with at least one record. For instance, we could create a file '`names.dat`' containing

```
Harry   S       Truman  88
```

where a single tab character separates the words. It doesn't matter whether the file contains a final newline. You don't have to type this in, because files '`names.dat`' and '`names.fmt`' are provided in the EDB examples distribution (see section "Installation" in *EDB Installation Manual*).

The easiest way to create a database file laid out in EDB's internal representation, which permits faster reading and writing of the data file, is the following. Create a database in some simpler format (such as tab-delimited text), read it in the usual way, set database slot `internal-file-layout-p` to `t` (for instance, via `C-u M-x db-toggle-internal-file-layout RET`), and finally write the database to disk via `C-x C-w`. It will be saved in EDB's internal file layout; for details, see Section 13.1 [Internal file layout], page 43. For more information about different data file layouts, see Chapter 13 [Database file layout], page 43.

The format file specifies how a record is displayed on the screen; a backslash followed by a field name indicates that the field's contents should be inserted there. For instance, here is a format for our names database:

```
Family: \last
Given:  \first
Middle: \middle

Age:    \age
```

For more information about specifying display formats, see Chapter 14 [How information is displayed], page 58.

The auxiliary file is used to customize EDB for a particular database; the only information that it absolutely must contain is the names (and possibly types) of the database fields. They are set by function `database-set-fieldnames-to-list` (see Section 16.1 [The database structure], page 79); by providing additional information to this call, you can specify types other than string (which is the default) for your record fields. Any auxiliary file customization may instead appear in the "Local variables" section of the format file (see Section 13.5 [Reading from disk], page 55). So our format file '`names.fmt`' would look like this:

```
Family: \last
Given:  \first
Middle: \middle

Age:    \age
```

```
Local Variables:
eval: (database-set-fieldnames-to-list database
                                       '(first middle last (age . integer)))
End:
```

For more information about customizing EDB for your application, see Chapter 15 [Customization], page 68.

Now that we have the two files '`names.dat`' and '`names.fmt`', we are ready to invoke the database. Simply type `M-x db-find-file RET names.dat RET`; '`names.fmt`' will be automatically used as the format file, and you can begin editing the database and adding new records.

11.2 Manipulating database fields

A future version of EDB may have a graphical method to manipulate databases by adding, removing, and rearranging fields, among other other manipulations; a rudimentary version of this functionality exists in the '`db-convert.el`' file but lacks a good user interface or documentation. Here is a way to perform those actions by directly manipulating the database file.

Make sure the database is stored in a delimited layout (the rest of this section assumes that the standard tab-separated text layout is used; see Section 13.2 [Delimited file layout], page 44). If the database file is in EDB's internal file layout, convert it to a delimited layout by reading in the database, setting database slot `internal-file-layout-p` to `nil` (for instance, via `C-u 0 M-x db-toggle-internal-file-layout RET`), and saving the database (see Section 13.1 [Internal file layout], page 43). (If the database's `wrfr` slot is set, the result will not necessarily be delimited layout, but whatever that function specifies.)

In the tab-separated database format, you can add new fields by adding new tabs in the right places in each record; simply edit the database file to add the next fields as desired. Similarly, you can delete or rearrange fields. You may wish to use keyboard macros, or write an Emacs Lisp function, to help you with this chore.

When you perform this edit, be sure that auto-fill-mode is turned off, lest spurious newlines be added to the file.

You will also need to modify any parts of your database that depend on the number or order of fields; for instance, a call to `database-set-fieldnames-to-list` will need to be changed, and the format file(s) should be edited if you wish to be able to view or edit the contents of the new fields.

12 Record field types

All the records of a database contain fields of the same types: the fifth field might always contain an invoice number, for instance, but the invoice number would vary from record to record. This chapter describes how to specify and use different record field types.

The most important information about a record field is

- display representation onscreen and in reports
- EDB's internal representation
- storage representation in data files
- how to convert among these representations
- how to sort items of that type

This information is separated into a display specification and a record field specification. The display specification determines how a field's contents are displayed and parsed onscreen (say, in a data display buffer). The record field specification controls everything else about the record field; its information does not depend on the onscreen (or in-report) visual appearance of the field. The database designer specifies a displaytype for each display specification in the format file (that is, for each location in the data display buffer that will contain a representation of some record field). The database designer specifies a record field type for each field in a database record, whether or not the field is ever displayed. (A particular field's contents may appear zero, one, or more times in a data display buffer; one displayspec structure is created for each occurrence.)

Displaytypes and record field types are distinct; they supply complementary information. There is not even a one-to-one relation between them. A particular record field type may be displayed in any of a number of ways by using different displaytypes—dates are such an example. On the other hand, record field types which are interpreted, sorted, and stored on disk differently, but which have the same internal representation—say, as a string—could all be displayed and edited using the same displaytype.

This chapter discusses record field types, record field specifications, and the recordfieldspec structure, the internal structure which holds the information specified by the former two items. For more information about displaytypes, display specifications, and the displayspec structure, see Chapter 14 [How information is displayed], page 58.

12.1 Specifying a record field type

The function `database-set-fieldnames-to-list` (see Section 16.1 [The database structure], page 79) specifies the types (and names) of record fields.

A record field type gives information about one field of the database's records: the type of the contents, what sorting function to use, how to write it to disk and read it back, constraints on its value, etc. The database designer must provide one for each record field. Usually one of the predefined types (see Section 12.2 [Predefined record field types], page 39) suffices. The remainder of this section describes how to define a new record field type, when that is necessary.

The first step in creating a new record field type is to make a recordfieldspec structure (when a predefined record field type is used, EDB looks up a built-in recordfieldspec). A recordfieldspec can be created from scratch by calling `make-recordfieldspec`, but it is often easier to modify a copy of an existing one. (For a list of built-in record field types, see Section 12.2 [Predefined record field types], page 39.)

`recordfieldtype->recordfieldspec`
: Return the recordfieldspec associated with symbol *recordfieldtype*.

`copy-recordfieldspec`
: Not documented.

Next, set the recordfieldspec's slots to appropriate values by using `recordfieldspec-set-`*slotname*; for a list of the slots, see Section 12.3 [The recordfieldspec structure], page 40. Finally, install the record field type by calling `define-recordfieldtype-from-recordfieldspec`:

`define-recordfieldtype-from-recordfieldspec`
: Define a recordfieldtype named *typename* (a symbol) with the default *recordfieldspec*. DISPLAYSPEC may also be a typename symbol itself. After this call, `recordfieldtype->recordfieldspec` called with argument *typename* returns the proper record field specification.

Examples of record field type creation can be found in the file '`db-types.el`', which contains a number of record field type (and displaytype) definitions that can be studied or copied.

Record field types should not be confused with display types; a display type is used to specify how a particular value is shown on the screen, but a record field type constrains the information actually contained in the record field.

12.2 Predefined record field types

The following record field types are predefined by EDB; their definitions can be found in the file '`db-types.el`'. The recordfieldspec structure contains a record field type's information; see Section 12.3 [The recordfieldspec structure], page 40. You can define record field types in the same way as '`db-types.el`' does; these record field types are not privileged in any way and are provided only for convenience. For more information about recordfieldspec creation, see Section 12.1 [Specifying a record field type], page 38.

integer Ordinary integers.

integer-or-nil
: Integers or `nil`, the empty value; by default, `nil` is treated as larger than any integer, so it comes last in an increasing-order sort.

number Ordinary numbers. A number is either an integer or a floating-point number.

number-or-nil
: Numbers or `nil`, the empty value; by default, `nil` is treated as larger than any number, so it comes last in an increasing-order sort.

boolean This displayspec corresponds to the yes-no displaytype. For the purposes of sorting, `t` is considered less than `nil`, so it appears first in an increasing-order sort.

string Ordinary strings.

one-line-string
: Strings which may not contain newlines.

string-or-nil
: Either a string or the value `nil`, which is converted to the empty string. Sorting treats `nil` identically to the empty string.

nil-or-string
: Identical to the string-or-nil recordfieldspec (except for the name). This exists so that display fields of type nil-or-string can conveniently default to this recordfieldspec.

one-line-string-or-nil
: The obvious combination of the one-line-string and string-or-nil recordfieldspecs.

date A date which specifies zero or more of the year, month, and day. By default the date is sorted by year, then month, then day; an unspecified component is larger than any specified component ("March 14, 1967" would appear before "January 1" if dates were sorted in increasing order). Dates are read from database files using the function value of `storage-string->date`, which is set by default to `date-stored->actual`, which can parse nearly any string representation of a date and returns a date if it is passed one.

Dates are written using the function value of `storage-string->date`, which defaults to `format-date-full`, which produces a string of the form "March 14, 1967". If the speed of reading and writing database files is very important to you, consider using `fset` to set `date->storage-string` and `storage-string->date` to more efficient functions, such as `date->storage-string-mmddyyyy` and `storage-string-mmddyyyy->date`, or `date->storage-string-lisp` and `storage-string-lisp->date`.

date-or-nil A date, or nil.

date-efficient-storage
When the dates in a database file are known to have a particular format, using `parse-date-string` is unnecessarily inefficient. The date-efficient-storage recordfieldspec specifies that `storage-string->date`, which can efficiently read dates written by `simple-format-date`, be used instead. The time savings is noticeable on large databases.

time A time.

If you find any of these typenames cumbersome, you can create your own aliases for them using `define-displaytype-from-displayspec`, `define-recordfieldtype-from-recordfieldspec`, or `define-type-alias`.

`define-type-alias`
Make symbol *alias* refer to the same displaytype and recordfieldtype as *typename*.

12.3 The recordfieldspec structure

The recordfieldspec structure contains information regarding the content of a record field, but nothing concerning how it is displayed on the screen or read from user input.

The slots of a recordfieldspec are listed below; a slot may be accessed by using the macro `recordfieldspec-`*slotname* and set using the macro `recordfieldspec-set-`*slotname*, whose second argument is the value to be stored in the slot.

Most of the slots may be left `nil`, and reasonable default actions will occur.

`type` A symbol such as `string` or `integer`, the type of the data described by this recordfieldspec.
When no displaytype is explicitly specified in a display specification, then a displaytype with the same name as `type` slot is used by default; this is the only use for this slot.

`default-value`
: The default value for fields described by this recordfieldspec; used when creating new records.

`common-form-function`
: A function which, called on the contents of a record field, returns them in canonical form. This can be used for determining non-trivial equality, when two nonidentical values should be considered equivalent.

`merge-function`
: A function which, called on the contents of two record fields, returns a combination of the two. Often it queries the user for help.

`order-function`
`sort-function`
: The record field's ordering and sorting functions (see Section 6.1 [Sorting functions], page 21). Both ordering and sorting are possible if either slot is filled.

  If both slots are empty, then a dummy ordering or sorting function is used, so sorting on this field has no effect. Since the function is called and its result examined, this is more expensive than not sorting on the field in the first place. If it does not make sense to sort on a particular field, then it is best to keep that field out of the field priorities used for sorting (which is the `field-priorities` database slot, or is interactively specified through the database sort interface (see Chapter 6 [Sorting], page 19), or is specified as an argument to `database-sort`).

  Actually, the recordfieldspec structure has no order-function or sort-function slots; they are fictions built upon the (real) order-fn and sort-fn slots. The user should not attempt to use those slots directly, but should always use the order-function and sort-function pseudo-slots.

`match-function`
: A function which takes a pattern and a field value and returns non-`nil` if they match. The function should also be able to take as its first argument a field value rather than a pattern.

`help-info`
: A string which is displayed by `db-field-help` when there is no field-specific help available. Field-specific help is usually preferable to this help, which only describes the type of the field's contents.

`actual->stored`
: A function which converts a field value into its on-disk representation (a string).

`stored->actual`
: A function which recovers a field value from its on-disk representation (a string). If this function returns a string, it should return something reasonable if supplied the empty string as its argument. (That can happen when an empty database is read.)

`change-hook`

A function called when the value of this field is changed. This is not currently implemented. For more change hooks, see Section 15.2.5 [Display format change hooks], page 73, and see Section 15.2.6 [Recordfieldspec change hooks], page 76.

`constraint-function`

A function which the value of this field must satisfy; that is, the function must return non-`nil` on it. The function may reject the value either by returning `nil` or by signalling an error; the latter permits the function to provide an informative message about the problem.

Four arguments are supplied to `constraint-function`: the field value, the record, the record fieldnumber, and the database. This permits cross-field and cross-record constraints. The record argument may be `nil`, in which case the function should return `t` if the value is acceptable for some conceivable record. This occurs, for instance, when values are read in a call to `db-field-query-replace`.

The constraint function may interact with the user; for instance, it may give you the opportunity to override the constraint.

13 Database file layout

This chapter discusses specifying how a database is read from a file (or saved back to it).

Broadly stated, there are three possible file representations for a database: EDB's internal file layout, a delimited layout, or a nonregular layout. EDB's internal file layout is designed for fast reading and writing, but is not very human-readable. A delimited layout is one in which records (and fields within a record) are separated from one another in a predictable (though not necessarily invariant) way. A nonregular layout is any other kind of layout; you may specify arbitrary Emacs Lisp code to read and write such files. EDB also supports tagged file layout (a special case of nonregular file layout).

If the database is to be stored in EDB internal file layout, a lot of this information is not needed except when the database is first created.

The sections of this chapter each describe a file layout, except the last, which describes in detail the process of reading a database from disk.

13.1 Internal file layout

The first line of a database file in EDB's internal file layout looks something like

```
;; Database file written by EDB; format 0.5
```

followed by two Emacs Lisp forms: a record (the database structure) and a list of records (the records of the database). Databases stored in this layout can be loaded and saved very quickly (sometimes orders of magnitude faster than databases which EDB must parse when reading), and they never suffer from ambiguities between data and delimiting text, but they are not easy for people to read and understand. A human- or program-readable version of the database can be generated when it is needed, either by creating a report or by saving in some other file layout. This is a good option when all manipulation of a database will be done via EDB.

Since this file layout is rather complicated, databases are often created in some other file layout and then converted to this one. To convert from another file layout to EDB's internal file layout, read in the database, set database slot `internal-file-layout-p` to `t` (for instance, via `C-u M-x db-toggle-internal-file-layout RET`), and then write or save the database (via `C-x C-w` or `C-x C-s`). Converting a database from EDB file layout to some other representation is similar, but slot

`internal-file-layout-p` is set to `nil` (say, via `C-u 0 M-x db-toggle-internal-file-layout RET`), and certain variables and database values may need to be set (see the documentation for the layout you desire, elsewhere in this chapter). Making a report (see Chapter 9 [Reports], page 28) can also produce a different file layout for a database, with even more flexibility than the techniques described here.

A database file in EDB internal file layout is basically the printed representation of the Lisp database structure used by EDB when the database is read in. As such, it contains all the information in the database slots described in Section 16.1 [The database structure], page 79, except that the `data-display-buffers` and `first-link` slots are set to `nil`; in the file, the records follow the database structure. After a database has been saved in internal file layout, then any forms in the auxiliary or format files that set database slots can be removed if desired; this is not necessary, however.

13.2 Delimited file layout

EDB can conveniently read and write database files in which records are separated from one another by a record delimiter and, within each record, fields are separated by a field delimiter. When the delimiters are the newline and tab characters, respectively, the result is the standard "tab-separated text" layout, which is often used for transferring information from one program to another.

The record and field delimiters need not be single characters; they can be arbitrary strings or even regular expressions. The latter is useful if the exact delimiter is not known ahead of time (for example, if records may be separated by one or more carriage returns). This regular expression mechanism can only be used when reading the database: when writing a database, all the record delimiters are identical, and so are all field delimiters. (Exception: you may specify an arbitrary record-writing function (see Section 13.4 [Nonregular file layout], page 52) and arbitrary functions for either reading records or for separating records or fields in delimited layout, but should use the simpler reading mechanisms whenever possible, for your own sake.)

Delimited file layout has two disadvantages. First, it is somewhat slower to read and write than EDB's internal representation. Second, the strings used as delimiters may not appear in the database fields, lest those occurrences be misinterpreted as delimiters rather than as data. For solutions to the latter problem, See Section 13.2.2 [Resolving ambiguities], page 48.

13.2.1 How to specify delimited file layouts

In a database stored in delimited file layout, records and fields can be separated by particular strings, by regular expressions, by context-sensitive regular expressions, or by arbitrary functions. The sepinfo structure holds this information for use when reading the database from disk (and writing it back). The sepinfos used when reading a database are stored in its `record-sepinfo` and `field-sepinfo` slots (for more details about the database structure, see Section 16.1 [The database structure], page 79).

When reading, if a separation function is specified, it is used; otherwise, if a regular expression is specified, it is used; otherwise, a string must be specified. It is converted into a regular expression, the regular expression slots of the sepinfo are filled in, and reading proceeds as if you had specified a regular expression. (To have a regular expression recomputed when it is next needed, set it to `nil` when setting the corresponding string value.)

When a sepinfo is used for writing, it must specify literal string separators. (The sepinfo may have its separation function or regular expressions set as well, for reading, but those slots are ignored when writing.)

13.2.1.1 The sepinfo structure

The sepinfo structure contains the information required to decide where records or fields start and end (actually, to determine where the record or field separators start and end; "sepinfo" is short for "separator information"). The slots of this structure may be accessed by using the macro `sepinfo-`*slotname*. The slots may be set using the macro `sepinfo-set-`*slotname*, whose second argument is the value to be stored in the slot.

The `pre-first-` slots describe text that precedes the first item of interest. In a record sepinfo they describe the file header, which precedes the first record. In a field sepinfo they describe any information that preceded the first field of every record, after the record separator.

The `post-last-` slots are similar, but are used to inform EDB of text following the last information-carrying text. In a record sepinfo, they describe the file trailer, which follows the last record in the file. A field sepinfo's `post-last-` slots tell about information following the last field of a record but preceding the record separator.

The `-submatch` integers describe which submatch of a regexp match is the actual separator, as opposed to surrounding text used to help make the match unambiguous. This specification of

the submatch permits context-sensitive matching that you might otherwise expect could not be done with regular expressions alone. For instance, suppose a database has records with a variable number of fields separated by newlines, that records are also separated by newlines, and that the first field of each record has some special form different from all other fields (say, it is a number with a decimal part). The following code would permit separation of the records without writing a special function to do so and without including the decimal number in the separating text:

```
(sepinfo-set-sep-regexp (database-record-sepinfo database)
                        "\\(\n\\)[0-9]+\\.[0-9]+")
(sepinfo-set-sep-regexp-submatch (database-record-sepinfo database) 1)
```

When you set the slots of the sepinfo, be careful to use a correct value. For instance, if your record separator is a form feed on a line by itself, you probably want to set the `sep-string` slot of the database's `record-sepinfo` to '`"\f\n"`', or possibly '`"\n\f\n"`', rather than just '`"\f"`', lest the newlines be considered to be part of the records rather than part of the separator.

The slots of the sepinfo are listed below but are not described in detail; see the preceding description for details of their use.

`pre-first-string`
: Setting the slot to `nil` (or not setting it) is equivalent to setting it to the empty string.

`pre-first-regexp`

`pre-first-regexp-submatch`

`sep-string`

`sep-regexp`

`sep-regexp-submatch`

`sep-function`
: A function that takes a buffer position, the end of the previous separator (that is, the start of the current record or field), as its argument and returns a pair of two buffer positions bracketing the next separator. That is, the returned values are the end of the current record or field and the beginning of the next one (or nil if there are no more). When the function is called, point is at the beginning of an item and the buffer is narrowed to the list being currently processed.

  The use of a separation function is useful when the separation criteria cannot be expressed as a combination of context-sensitive regular expressions. The `pre-first-` and `post-last-` slots are still used even if a function is specified.

`post-last-string`
: Setting the slot to `nil` (or not setting it) is equivalent to setting it to the empty string.

`post-last-regexp`

`post-last-regexp-submatch`

13.2.1.2 Record and field separator examples

[[[Put examples here.]]]

[[[For instance, to parse "[Mary, John,Jack, and Jill]" and to write it back out as "[Mary, John, Jack, Jill]", the following specification would suffice: pre-first-string "[" sep-string ", " sep-regexp ", +\\(and +\\)?" sep-regexp-submatch 0 post-last-string "]"]]]

[[[The -string slots are used for writing; but what if you only have a regexp for the leading or trailing junk, but you want that restored exactly? You can set pre-first-string *after* the database file has been found. For instance, in db-before-read-hooks, use a function such as

```
(defun btxdb:read-comments ()
  (save-excursion
    (set-buffer db-buffer)
    (goto-char (point-min))
    (if (search-forward "@" nil t)
        (sepinfo-set-pre-first-string
         (database-record-sepinfo database)
         (buffer-substring (point-min) (point))))))
```

or even put

```
(sepinfo-set-post-last-string
 (database-record-sepinfo database)
 (save-excursion
   (set-buffer db-buffer)
   (goto-char (point-min))
   (re-search-forward "\n\C-l\n")
   (buffer-substring (match-beginning 0) (point-max))))
```

as is in your auxiliary file.]]]

[[[If all records have the same number of lines on disk, use the following function to return an appropriate sep-function. This is useful when, for instance, both the field separator and the record separator are the newline character.

`make-n-line-sep-function`
: Return a sep-function useful when all records have exactly *n* lines on disk.

]]]

13.2.2 Resolving ambiguities

Substitution is a mechanism for dealing with the problem of distinguishing field and record separators from the contents of database records. For instance, if the newline character (actually, a string consisting of only the newline character) is used as a record separator, and records may contain multiline text fields (or other fields whose storage representation contains a newline), then how would EDB know, when reading the database back in, which newlines are record separators and which are part of fields?

There are several ways to avoid this ambiguity.

- Disallow the use in record fields of the character or string causing the ambiguity. For instance, in the example above, you might change the record field type of all of the string fields to one-line-string.
- Change the separator(s) to strings that do not appear in the storage representation of any field. For instance, Unix password files are stored in delimited file layout with a colon as the field delimiter (and colons are prohibited from appearing in the field text).

  ```
  (sepinfo-set-sep-string (database-field-sepinfo database) ":")
  ```

  Strings containing non-printing characters are another good bet, but this method relies on luck and the hope that the chosen separators will never appear in data.
- Change the representation of the ambiguous string, when it appears in data; this guarantees that whenever the string does appear in a database file, it stands for a separator. This scheme is called substitution, because another string is substituted for the ambiguous one when it appears in data. This is similar to the previous workaround, which changed the separators rather than the data-bearing instances of the string. Ambiguities are still possible, if the substituted text happens to appear elsewhere in data. Specifying a substitution is described below.
- The simplest solution is to use EDB's internal file layout (see Section 13.1 [Internal file layout], page 43). Ambiguities can only occur when the field data and the separators are both text to be interpreted by EDB. EDB's internal file layout uses Emacs Lisp's mechanisms (a built-in form of quoting) to ensure that what is read in is identical to what was written out. The database designer need not worry further about the problem.

Substitution is the replacement of potentially ambiguous strings by other ones. For instance, when writing tab-separated text, each occurrence of the newline character in a field could be replaced by control-k when the database is written. Then, when the file is read in, every newline can be safely assumed to be a record separator. The final step is converting the control-k characters back into newlines. This approach is taken by some marketed databases; for instance, I believe that FileMaker does just this. The problem with this approach is that if there were any control-k characters in the text, then when the database is read back in, they will be (incorrectly) converted to newlines. EDB warns when the database is being written out if this problem could occur; you may choose a different substitution or abort the database write operation. It is usually possible to find a substitution—a character or sequence of characters that doesn't appear in the data.

The database's substitutions slot is an association list of data strings and their file representations. To make control-k in the database file stand for newline in the data, put

```
eval: (database-set-substitutions database '(("\n" . "\C-k")))
```

in the "Local Variables" section of your format file, or put the form without '`eval:`' in the auxiliary file.

13.2.3 Problems with end-of-file newlines

Suppose you want to get rid of every newline at the end of the database file, but you don't know how many there are.

```
(sepinfo-set-post-last-regexp (database-record-sepinfo database) "\n*\\'")
(sepinfo-set-post-last-regexp-submatch (database-record-sepinfo database) 0)
```

does not work, because the post-last-record regexp is searched for backward from the end of the buffer, and (because of the way that `regexp-search-backward` is implemented) the backwards regexp match for '`\n*`' is always the empty string! The proper way to write this is

```
(sepinfo-set-post-last-regexp (database-record-sepinfo database)
      "[^\n]\\(\n*\\'\\)")
(sepinfo-set-post-last-regexp-submatch (database-record-sepinfo database) 1)
```

13.3 Tagged file layout

Another popular file layout supported by EDB is that of field values preceded by the fieldname. For instance, a record might be represented in the file by

```
Where:Here
When:  Now
What: This!
```

which indicates a record in which the 'where', 'when', and 'here' fields have the specified values.

Tagged files are a special case of files in nonregular layout; support for them is implemented through the mechanisms described in Section 13.4 [Nonregular file layout], page 52.

To read a database file in tagged format, call the function `db-tagged-setup` in the database's format or auxiliary file. Its argument specifies the names of the fields and the tags that precede them in the database file.

`db-tagged-setup`

> Ready the database to read files in tagged format. Creates database local variables and sets database slots. Argument *tagged-field-specs* is a list of tagged-field specifications, one for each field in a database record. Each tagged-field specification is a three-element list of the field name (a symbol), the tag used to identify it in the file (a string), and a brief help string. Instead of a symbol, the tagged-field name may be a cons of the field name and its type. To indicate that a field is never found in the input file (typically because it is computed on the fly), use `nil` for its tag.
>
> This function should be called first in an auxiliary or format file, so that the defaults it chooses can be overridden. `database-set-fieldnames-to-list` should not be called if this function is.

Calling `db-tagged-setup` sets the database's field names and installs appropriate functions for reading and writing the database. It also creates some database-local variables (see Section 15.3 [Local variables], page 76) which can be modified (by use of the `database-set-local` function) in order to customize the behavior of the parsing and output functions with respect to what characters can appear in a tag, what the separator between tag and value looks like, and how continuation lines are handled. By default, records are separated by blank lines, tags are separated from field values by ':', white space around the separator is not significant on input, the separator is followed by one tab on output, and continuation lines start with whitespace.

`db-tagged-tag-chars`
: The characters that are allowed in field tags, in a form suitable for placing inside [] in a regular expression.

`db-tagged-separator`
: The string that separates field names from values. Used only if `db-tagged-separator-regexp` or `db-tagged-separator-output` is `nil` (depending on whether the record is being read or written).

`db-tagged-separator-regexp`
: A regexp for the separator between field names and values when parsing.

`db-tagged-separator-output`
: The separator between field names and values on output.

`db-tagged-continuation`
: The string that marks (the beginning of) a continuation line. Used only if `db-tagged-continuation-regexp` or `db-tagged-continuation-output` is `nil` (depending on whether the record is being read or written).

`db-tagged-continuation-regexp`
: A regexp for a continuation line in a value when parsing.

`db-tagged-continuation-output`
: The fixed string to use (before) continuing values on output.

Other hooks permit arbitrary manipulations of records; for instance, if a database nearly conforms to the tagged file model, these can be used to customize the behavior of the existing tagged code. One way to do this is to have a function in `db-tagged-rrfr-hooks` remove the field from the file representation before the record is parsed, then have `db-tagged-wrfr-after-hooks` modify the automatically generated tagged file representation for that field. These functions can also be used for simpler tasks, of course.

`db-tagged-rrfr-hooks`
: Hooks run on each database record before tagged parse.

`db-tagged-wrfr-before-hooks`
: Hooks run before each tagged write of a database record. The record is bound to the dynamic variable record, and point is where the record will be inserted in the buffer.

`db-tagged-wrfr-after-hooks`
: Hooks run after each tagged write of a database record. The record is bound to the dynamic variable record, and point is immediately after the file representation of the record.

13.4 Nonregular file layout

Unlike most databases, EDB can work with data stored in any file layout whatever—so long as you specify how the information is to be extracted. If the file layout is too complicated to be described by regular expressions describing the record and field separators and their context (see Section 13.2 [Delimited file layout], page 44), then you may write Emacs Lisp code which extracts the information from the database file.

The great advantage of this mechanism is that it permits you to maintain your current files, in exactly their current file layouts, and to keep the same tools and habits you've accumulated, but also to manipulate them in a structured way with EDB. For instance, you might wish to maintain the database file in a file format easy for people to read all the time, rather than having to create a report for that purpose.

Three pieces of information must be provided: how to find the extent of a file record, how to read a file record, and how to write a file record. The third may be omitted if the database is only being read in the custom file layout (and will be saved in some more tractable file format). If the second is provided (that is, the `read-record-from-region` database slot is set), then the file will be assumed to be in a nonregular file layout and the value of that slot is used to read the database, no matter what other information is provided.

Information about how to separate one record from another within the file is found in the `record-sepinfo` slot of the database, as usual. In many cases, even if the file layout of the data is nonregular, it is easiest to describe the record separator with a string or a regexp. For more details, see Section 13.2 [Delimited file layout], page 44. You may also set the sepinfo's `sep-function` slot to a function. The function should take one argument, the end of the previous record (`nil` the first time it's called), and return a pair whose car is the end of the current record and whose cdr is the start of the next record (`nil` if there is no next record in the file).

The `read-record-from-region` slot of the database contains a function of no arguments which, when called with the current buffer narrowed to a single file record (that is, narrowed to the representation of a single database record), returns a record in the database's internal file layout. The variable `database` is dynamically bound to the current database, and so the right way to create the record to be returned is via `(make-record database)`. Its fields can then be set with `record-set-field`.

The `write-region-from-record` slot of the database optionally contains a function which takes a database record as its argument and inserts the file representation of that record in the current buffer; the variable `database` is dynamically bound to the current database. If this slot is not

specified (and slot `internal-file-layout-p` is `nil`), then the fieldsep and recordsep information, if present, is used to write the record (see Section 13.2 [Delimited file layout], page 44). This permits the use of a simple, delimited output file layout with a more flexible input file layout.

Tagged format is a special case of nonregular file layout for which EDB provides support; see the implementation of support for tagged database files in '`db-tagged.el`' and Section 13.3 [Tagged file layout], page 50. Another example is given below.

13.4.1 Example of database in nonregular file layout

Here is a simple example of a database in a nonregular file layout; this does not mean that the file representation of each record is vastly different from the others (it may be, but is not in this instance), but that there is no regular rule for extracting field values from the record.

Suppose we had a database with fields '`place`', '`time`', and '`purpose`', whose database file was:

```
Dentist's Office at Never! for Root canal
Home at Midnight for Sleep
Other places at Other times for Other things
```

In order to read and write this database, place the following code in the auxiliary file (see Section 13.5 [Reading from disk], page 55):

```
(database-set-read-record-from-region database 'arb-demo-rrfr)
(database-set-write-region-from-record database 'arb-demo-wrfr)

(defun arb-demo-rrfr ()
  (goto-char (point-min))
  (if (re-search-forward
       "\\(.*\\)\s-+at\\s-+\\(.*\\)\s-+for\\s-+\\(.*\\)")
      (let ((result-record (make-record database)))
        (record-set-field result-record 'place (db-match-string 1) database)
        (record-set-field result-record 'time (db-match-string 2) database)
        (record-set-field result-record 'purpose (db-match-string 3) database)
        result-record)
    (error "This didn't look right to me.")))

(defun arb-demo-wrfr (record)
  (insert (record-field record 'place database)
          " at " (record-field record 'time database)
          " for " (record-field record 'purpose database)))
```

The auxiliary file would also specify the database's fieldnames:

```
(database-set-fieldnames-to-list database '(place time purpose))
```

as well as possibly other information such as the summary format or the name of the default format file. See the example database auxiliary file '`arb-demo.dba`' for a concrete example of this.

All this Emacs Lisp code may be placed in "Local Variables" section of the format file instead of in the auxiliary file, if desired. For more information about the "Local Variables" section of a file, See section "Variables" in *The GNU Emacs Manual.*

This particular example is simple enough that a special function for reading isn't strictly necessary. Reading can be done under the control of regular expressions; for instance, each field separator would be `"\\s-+\\(at\\|for\\)\\s-+"`. See the example database auxiliary file '`arb-demo-regexp.dba`' for a concrete example of this. You would still need to specify a special record-writing function.

Here is another example, which has no field separators; in the data file, the fields abut one another. While it, too, could be read and written under the control of regular expressions, the use of functions is a bit clearer.

The data file is:

```
Einsteinbirthday03141879
Botswanaindepend09301966
```

The auxiliary file contains the following:

```
(database-set-print-name database "Historic Dates")
(database-set-fieldnames-to-list database '(name occasion month day year))

(sepinfo-set-sep-string (database-record-sepinfo database) "\n")
(database-set-read-record-from-region database 'sized-field-rrfr)
(database-set-write-region-from-record database 'sized-field-wrfr)

(defvar sized-field-alist
  '((name . 8)
    (occasion . 8)
    (month . 2)
    (day . 2)
    (year . 4)))
```

```
(defun sized-field-rrfr ()
  (goto-char (point-min))
  (let ((result-record (make-record database))
        (field-begin (point)))
    (mapcar (function
             (lambda (this-size-cons)
               (forward-char (cdr this-size-cons))
               (record-set-field result-record (car this-size-cons)
                                 (buffer-substring field-begin (point))
                                 database)
               (setq field-begin (point))))
            sized-field-alist)
    (if (not (eobp))
        (error "Found extra characters in this record."))
    result-record))

(defun sized-field-wrfr (record)
  (mapcar (function
           (lambda (this-size-cons)
             (let ((this-field-value (record-field record
                                                   (car this-size-cons)
                                                   database)))
               (if (not (= (length this-field-value) (cdr this-size-cons)))
                   (error "Field %s value \"%s\" should have length %d, not %d."
                          this-field-value (car this-size-cons)
                          (cdr this-size-cons) (length this-field-value)))
               (insert this-field-value))))
          sized-field-alist))
```

This example is actually a bit too simple. Some of the fields could be made non-strings, field constraints should keep the fields the right length, and `sized-field-alist` should be a database-local variable.

13.5 What happens when a database is read in from disk

In brief, the following happens after you execute `db-find-file`:

1. If the database is already read in and its buffer has not been killed, the buffer is simply selected. No other work is done.
2. Otherwise, the database file is inserted in a special buffer of its own. If the database is in EDB internal file layout (that is, if an identifying header is found), it is read in immediately. Otherwise, a new, empty database is created. In either case the dynamic variable `database` is

bound; this makes it possible to refer to the database in the auxiliary and format files (even before it has been read in, if it is not in EDB internal file layout).

3. The format file is found (see Section 15.1 [Auxiliary files], page 68), and the data display buffer is created.
4. The function `db-setup-data-display-buffer` is called; it performs the rest of the work necessary for setting up the data display buffer (everything up to the running of `db-before-read-hooks`). Its first action is to insert the format file's contents into the data display buffer.
5. The auxiliary file, if any, is loaded. This happens in the data display buffer, and the dynamic variable `database` and the buffer-local variable `dbc-database` are bound to the current database. For more information about how the auxiliary file is found and what it can do, see Section 15.1 [Auxiliary files], page 68.

   The auxiliary file is not read every time `db-setup-data-display-buffer` is called, only when a database's primary display format is read. (The primary display format is the one initially selected when a database is first read in.)
6. The "Local Variables" section, if any, of the format file is executed; this may set variables and execute Emacs Lisp code, exactly analogously to the auxiliary file. EDB ignores the value of `inhibit-local-variables` when evaluating this code. This section is then deleted from the working copy of the file, so that it does not appear in the data display buffer when you view database records. For more information about the "Local Variables" section of a file, see section "Variables" in *The GNU Emacs Manual*.
7. Database information is propagated; for instance, the names of the database fields are known by now, and various other database slots are filled in depending on this information, if they haven't been set yet.
8. The format file is parsed, and literal text and formatting directives are distinguished from one another. This work is done by the `db-setup-format-parse-displayspecs` function. When that function is done, `db-setup-data-display-buffer` returns the database data display buffer as its result.
9. The hooks in `db-before-read-hooks` are run in the data display buffer.
10. If the database had already been read because it was stored in internal file layout, it is massaged a bit to get it into its final form; for instance, the backward links are added between adjacent records.

    Otherwise, the database is finally read; the values of the `recordsep` and `fieldsep` slots of the database determine whether the layout is delimited or nonregular and direct the parsing. The `substitutions` slot directs replacement of characters that could not be written into the file, and the `stored->actual` slot of each recordfieldspec completes the translation to the data's internal format from its file layout.
11. The hooks in `db-after-read-hooks` are run in the data display buffer.
12. The database has now been read and is in its final form. The first record of the database is

displayed in the data display buffer, which is then placed in view mode and selected (made visible).

14 How information is displayed

The display of information, both on the screen (whether in the data display buffer, the summary buffer, or elsewhere) and in other output (such as reports), is controlled by formatting commands. We will discuss a data display buffer by way of example; the formatting specifications are the same for summary buffers and reports as well.

Display types should not be confused with record field types; a display type is used to specify how a particular value is shown on the screen, but a record field type constrains the information actually contained in the record field. This chapter does not discuss record field specifications, which specify everything about a record field type except how it is displayed and parsed in output intended for humans to read. For more information about that, and about the distinction between record field types and displaytypes (the latter of which is described in this chapter), see Chapter 12 [Record field types], page 37.

A *display format* gives all of the information necessary to create a data display buffer; it consists of literal text that is displayed as is (and may not be edited by a user of the database) and of *display specification*s that instruct EDB how to display a particular field's contents. The display specifications do not appear in the data display buffer; they are replaced by fields' values, which may or may not be editable. An example of a display specification is '`\name,width=16`', which indicates that the '`name`' field of the database should be displayed (after being padded or truncated to exactly 16 characters).

When a format is first specified, it is parsed and the formatting information specified in the display specification strings is used to create a displayspec structure. Users should never have to manipulate displayspecs directly.

14.1 Display specifications

A *display specification* describes how a particular database record field appears in the data display buffer. An display specification consists of a backslash followed by a field name, plus perhaps some optional type and formatting information, plus optionally a backslash followed by a space. The items of extra formatting information must be separated from the field name and from each other by commas. No spaces or tabs may occur in a display specification. To specify a backslash which does not begin a display specification, but should appear in the data display buffer verbatim, precede it by another backslash.

Here is a (quite complicated) example display format:

```
\name,one-line-string,actual->display=upcase\ , \occupation'
Pay:     \\\salary,min-width=4:  too much!
Address: \address,indent is home sweet home
```

This display format is valid if the database contains fields called '`name`', '`occupation`', '`salary`', and '`address`'; any other fields are not displayed. Some typical records would be displayed like this:

```
JOHN DOE, butcher
Pay:     \  22:  too much!
Address: 123 Main St.
         Anyplace, USA is home sweet home

JANE ROE, baker
Pay:     \4444444:  too much!
Address: 675 Massachusetts Avenue is home sweet home
```

The optional information includes the type of this display field and formatting directives for it; if the type is present, then it must come first among the displayspec's optional specifications. Each optional parameter is preceded by a comma to separate it from the preceding one (or from the fieldname, for the first optional parameter). The optional information is typically of the form '*slotname*`=`*value*', which sets the specified slot to the given value, or '*slotsetter*', which sets some slot to a particular value. Explicitly specified formatting information overrides any defaults. For a list of slotnames and slotsetters, see Section 14.5 [Display specification optional parameters], page 64.

The display type can be specified by writing a typename (such as '`string`' or '`integer`') as the first optional parameter. A type abbreviation may be used instead of a typename; the defined type abbreviations are '`#`' for integer, '`$`' for number, '`"`' for string, and '`'`' for one-line-string.

The display type specifies default values for the display specification (actually for the displayspec structure, which is derived from the display specification). It is rarely necessary even to specify the displaytype—most display specifications consist of simply a backslash and a fieldname—since if the displaytype is omitted then a displaytype with the same name as the record field type (actually the `type` slot of the recordfieldspec) is used. This works because typically displaytypes and recordfieldtypes with the same names and complementary definitions are declared at the same time. The displaytype must be compatible with the record field type; it is an error to specify a displaytype of '`integer`' when the data is actually a string. `database-set-fieldnames-to-list` to specify recordfieldtypes; see Section 16.1 [The database structure], page 79.

A display specification abbreviation, which consists of an abbreviation name preceded by a backslash (and so looks like a simple display specification in which no optional information is specified), can be used instead of a standard display specification; Section 14.6 [Display specification abbreviations], page 67.

14.2 Predefined displaytypes

The file '`db-types.el`' defines the following displaytypes, corresponding recordfieldtypes, and some useful associated functions. You can also define displaytypes of your own; see Section 14.4 [Defining new displaytypes], page 63.

integer Ordinary integers.

integer-or-nil
: Integers or `nil`, the empty value; `nil` is formatted as the empty string.

number Ordinary numbers. A number is an integer or a floating-point number.

number-or-nil
: Numbers or `nil`, the empty value; `nil` is formatted as the empty string.

yes-no This displayspec corresponds to the boolean recordfieldtype. The field is three characters long and contains "Yes" or "No ".

string Ordinary strings. By default there is no maximum or minimum width or height, and subsequent lines are indented relative to the first character of the first line.

one-line-string
: Strings which may not contain newlines.

string-or-nil
: Either a string or the value `nil`, which is displayed as the empty string.

nil-or-string
: Either a string or the value `nil`. When you enter the empty string as the field value, or when a new record is created, the value `nil` is used in preference to the empty string.

one-line-string-or-nil
: Either the value `nil` or a string which may not contain newlines.

date A date which specifies zero or more of the year, month, and day. The date is formatted by `format-date` and parsed by `parse-date-string`; for details, see Section 14.2.1 [Date displaytype], page 61.

time A time which specifies zero or more of the hour, minute, and second. The time is formatted by `format-time` and parsed by `parse-time-string`; for details, see Section 14.2.2 [Time displaytype], page 61.

14.2.1 Date displaytype

EDB defines a date abstraction and a variety of useful operations upon it; the best way to learn about these features is to read '`db-time.el`'

This section provides more detail about the date displaytype. A date specifies a year, month, and day (all integers); any or all of these components may be hidden. Dates are created by the constructor `make-date` and a date's components are retrieved using the selectors `date-year`, `date-month`, and `date-day`.

`make-date`
Make an EDB date object with arguments *year month day*.

[[[Document parse-date-string, format-date, simple-format-date, def-xxx-type.]]]

[[[I added several useful (to me, anyway) displayspecs for various date types. These are meant to be used in a display spec, like: \datefield,date-mmddyy The displayspecs are implemented with similarly named formatting functions, which I also implemented. All of the new formatting functions are named format-date-XXX, where XXX are the various styles.]]]

14.2.2 Time displaytype

[[[Similarly to the above, for times.]]]

14.3 Enumeration types

An enumeration displaytype is used for fields whose values are one of a fixed set of alternatives. Each alternative may be a single character (say, '`M`' or '`F`' for gender) or specifiable by a single character (for example, if the first letters of the alternatives are unique); you need only type a single character in order to select one of the alternatives. Another possibility is that each alternative consists of an entire string entered with completion. (The string may consist of only a single character

if desired, but you must still type `RET` after entering the string.) The internal representation of the data—its recordtype—need have nothing to do with the way that the alternatives are specified. The next two sections describe the two types of enumeration displaytypes, which are nicknamed "one-char-enum" and (for the multicharacter alternative type) "enum".

14.3.1 One-character enumeration displaytypes

One-character enumeration displaytypes are not yet implemented.

14.3.2 Multi-character enumeration displaytypes

Multi-character enumeration displaytypes require you to enter an entire string in order to specify one of the alternatives. This typing may be done with completion in the minibuffer, which means that keys such as `TAB` and `?` complete a partly-entered choice or list the remaining possibilities. (For more about completion, see section "Completion" in *The GNU Emacs Manual*.) The internal, input, display, and file storage representations of the value may all be different. Multi-character enumeration displaytypes (also known as enum displaytypes) are created by calling the following function, which also creates a corresponding recordfieldtype.

`define-enum-type`

> Make *typename* (a symbol or string) an enumerated type. Both a displaytype and a recordfieldtype are created.
>
> *alternatives* is a list. Each alternative is a list of up to four components: the internal representation, any constant Lisp object, often a string; the input representation typed by the user to specify this alternative, a string or list of strings (for multiple input representations); the display representation, a string; and the file storage representation, a string.
>
> If the input representation is omitted and the internal representation is a string, that string is used. If the display representation is omitted, it defaults to the first input representation. The display representation is automatically also a valid input representation. If the file storage representation is omitted, it defaults to the display representation. If all the other components are omitted, the internal representation string may be used in place of a one-element list containing just it.
>
> Optional argument *optstring* is a displayspec option string.

When a record field's type is an enum type, both EDB and code written by the database designer may assume that the value in the record field is one of the valid representations. (Similarly, when

a field's type is string, EDB can assume that the field content is actually a string.) This means that the empty string, `nil`, and other special values must be specifically mentioned when the enumeration type is defined. Here is a way to define an enumeration type which is either a day of the week or the empty string:

```
(define-enum-type 'workday
  '("Monday" "Tuesday" "Wednesday" "Thursday" "Friday" ""))
```

If it is possible for the field value to be `nil` (but not the empty string) after reading the database, and `nil` should be displayed as '`Unknown`' (and that string parsed into a value of `nil`), the following definition suffices:

```
(define-enum-type 'workday
  '("Monday" "Tuesday" "Wednesday" "Thursday" "Friday" (nil "Unknown")))
```

14.4 Defining new displaytypes

When you are about to type a complicated display specification—or a simple one more than once—consider defining and using a displaytype instead. Displaytypes are more concise (and so less cumbersome and less error-prone), easier to change (since a change to the displaytype can affect every display specification that uses it), and clearer (since a descriptive typename makes immediately clear what the intention is). Furthermore, displaytypes can be built up incrementally, with each one making a few changes to those from which it inherits defaults.

There are two ways to define a new displaytype; each requires specifying the name of the displaytype and some formatting information to be associated with that displaytype.

The first method permits a displaytype to be specified by the optional part of a display specification, which is a string consisting of comma-delimited optional parameters. The first optional parameter may be a type, in which case the defaults for values not explicitly set in the other parameters are taken from that type.

`define-displaytype-from-optstring`
: Define a displaytype named *typename* according to *optstring*. *typename* is a symbol or string and *optstring* is the optional parameters part of a display specification string.

The second method is more useful for complicated displaytypes; it is also somewhat more efficient, which is why the file '`db-types.el`' uses it to define the standard predefined types. This

method is to create a displayspec or copy an existing one, then modify it as desired using the structure slot modifiers (i.e., `displayspec-set-`*slotname*; see Section 14.5 [Display specification optional parameters], page 64, for a list of slotnames), and then associate a typename with the displayspec. In fact, this is precisely what `define-displaytype-from-optstring` does.

`make-displayspec`
: Not documented.

`displaytype->displayspec`
: Return a copy of the displayspec corresponding to string or symbol *displaytype*. Return `nil` if there's no corresponding displayspec.

`define-displaytype-from-displayspec`
: Define a displaytype named *typename* (a symbol) with the default *displayspec*. *displayspec* may also be a typename symbol itself.

14.5 Display specification optional parameters

This section describes the display specification optional parameters, which correspond exactly to slots of the displayspec, EDB's internal representation of the display specification.

Optional display specification parameters are separated only by commas; display specifications never contain whitespace. These parameters are of two forms: slotsetters, which are a single word and set a slot to a particular value; and slot assigners, which are of the form '*slotname*`=`*value*' and set the slot to the value. Unless otherwise specified, each slot can be set by a slot assigner whose name is the same as that of the slot. An example of a display specification containing two optional parameters, one a slot assigner and one a slotsetter, is '`\name,width=16,unreachable`'.

Display specification fields are processed in order, so only the last instance of a particular parameter has any effect. Any explicitly specified parameter overrides defaults, values inferred from the type, or previously specified parameters.

If you find yourself repeatedly writing similar display specifications, or large, bulky display specifications, consider defining a new type to do some or all of the work for you; see Section 14.4 [Defining new displaytypes], page 63.

`record-index`
: This integer is the field index in a database record of the value formatted by this displayspec. This is set by looking up the fieldname part of the displayspec.

`indent` This boolean value determines whether the second and subsequent lines should align with the beginning of the first one or should be flush left, in column 0. It is set and unset with the `indent` and `noindent` slotsetters. The first of the following displays has indent set, and the second does not:

```
Name:    John Doe
Address: 123 Main St.
         Anyplace, USA
```

```
Name:    John Doe
Address: 123 Main St.
Anyplace, USA
```

This causes alignment of the first character of subsequent lines with the first character of the first line; it does not do anything clever with whitespace in the field value, nor does it align different lines differently.

`min-width`
`max-width`

These integers are the minimum and maximum widths which the display may occupy. If the formatted value is too short, the function in the `padding-action` slot is called to lengthen and/or justify it. If the formatted value is too long, the function in the `truncation-display-action` slot is called to shorten it; if that slot is empty, the field is simply truncated. The `width` slot assigner sets both the `min-width` and `max-width` displayspec slots. The `min-length`, `max-length`, and `length` slot specifiers are synonyms for the `min-width`, `max-width`, and `width` slot specifiers.

`min-height`
`max-height`

These are analogous to `min-width` and `max-width`, but for the number of lines occupied by the formatted value (actually, the number of newlines in the string, plus one). There is a `height` slot assigner which sets them both.

`truncation-display-action`

This function helps reduce the size of the formatted value when it is too large to fit in the specified displayspec size. It defaults to simply truncating the formatted field to the maximum permissible size. It may also be set with the `trunc-display` slot assigner. At present, this function is only called if the formatted value is too wide; there is no analogous function called when it is too tall.

`padding-action`

This function determines how a field that is too small for the displayspec (that is, the printed representation contains fewer characters than specified in the `min-width` slot) should be expanded to fit. The padding function takes three arguments: the minimum length, the unpadded display representation, and the length of that representation.

The `padding-action` slot may also be set to a cons of a padding character and a padding direction: `nil` for left-justification (padding on the right), and non-`nil` for

right-justification. (You cannot set the `padding-action` displayspec slot to a cons by using a display specification, since display specifications may not contain whitespace, so the easiest way to right-justify a single field is to use the `right-justify` slotsetter.) The default, which can also be obtained just by setting the slot to `nil`, is to pad on the right with space characters.

`actual->display`
`display->actual`

These functions convert between the data's internal representation and its displayed representation (a string). Other functions (such as those in the `truncation-display-action` and `padding-action` slots) may then be called on the result returned by the `actual->display` function. These slots may be set with the `a->d` and `d->a` slot assigners.

The `display->actual` function takes either one argument or four arguments: either just the field text or the field text, the previous field value, the record being operated upon, and the record fieldnumber of the field in question. EDB ascertains at runtime how many arguments the function should be applied to. The old field value is passed in case it contains hidden (undisplayed) attributes that need to be preserved across changes. The other two arguments permit a particular `display->actual` function to be used for more than one field of a record, allow the field text parse to depend on other record field values, and provide for other complicated needs. Most `display->actual` functions can be specified to take a single argument.

The `actual->display` function takes either one argument or three arguments: either just the field value or the field value, the record, and the record fieldnumber. EDB ascertains at runtime how many arguments the function should be applied to. The reasons the additional arguments may be specified are similar to those outlined above (for instance, to permit the displayed representation of a field to depend on other information in the record); most `actual->display` functions take just one argument—for instance, `upcase` is a valid `actual->display` function.

`match-actual->display`
`match-display->actual`

These functions are like `actual->display` and `display->actual`, but are only invoked when reading a displaying a search specification. If they are not specified (as is usually the case), then the ordinary (`match-`-less) versions are used for search specifications too. For instance, a string type can use `dbm-string-prefix-regexp` to specify a regular expression search rather than a substring search.

These slots should be set to symbols, not to functions proper; that is, to specify that function `foo` should be used, set the slot to `'foo`, not to `(function foo)`.

I don't know that these belong in the displayspec, but I don't quite know where they do belong.

`truncation-editing-action`
: This function specifies what to do when a field being edited is too large for the specified displayspec size; this action may be different from that taken when simply displaying the offending value. It may also be set with the `trunc-edit` slot assigner.

`reachablep`
: A Boolean value determining whether movement commands should skip this display field. The `reachable` and `unreachable` slotsetters are used to assign a value to this slot.

14.6 Display specification abbreviations

Complicated display specifications—those which specify more than a few optional parameters—can clutter the display format, keeping it from looking like the data display buffer will look when a database record is being displayed. You may tolerate the complicated display specification, define a new displaytype (which would permit the display specification to consist of just the fieldname and displaytype), or use a display specification abbreviation which is defined elsewhere in the format file. The display specification abbreviation is a symbol which expands to a full display specification; when a display specification consisting of only the abbreviation is encountered, the expansion is substituted and processing continues. (Actually, a displayspec corresponding to the expansion is used, but since displayspecs are immutable, this doesn't make a difference.) Display specification abbreviations can be much more concise than ordinary display specifications, which contain at least a fieldname and often other information to boot.

[[[How to define display specification abbreviations.]]] [[[Poorly-named variable dbf-fieldabbrevs controls this; it isn't getting set anywhere, though.]]]

15 Customization

15.1 Auxiliary and format files

A database designer can customize a database by providing code to be executed when the database is read in (see Section 13.5 [Reading from disk], page 55). The optional auxiliary file usually contains the code specific to a particular database, but the format file, which specifies the on-screen arrangement of fields of a record, can also contain such code.

Since the auxiliary file is read after the format file has been found but before it has been parsed, neither file can specify the other. The format file can, however, load arbitrary files, which is nearly as good as being able to specify an auxiliary file.

If the `aux-file` database slot does not specify the auxiliary file, EDB looks for a file with the same name as the database file, but ending with one of the suffixes in `db-aux-file-suffixes`.

`db-aux-file-suffixes`
: List of auxiliary file suffixes; the basename is that of the database file. The suffixes are tried in order; the default is (".dba" ".aux" "a"). The . that may precede the extension must be specified explicitly.

`db-aux-file-path`
: List of directories (strings) to search, in order, for auxiliary files not found in the directory with their associated databases.

The auxiliary file is evaluated in the data display buffer and so can set variables local to that buffer, such as hooks (see Section 15.2 [Hooks and customization functions], page 69). The database itself can be manipulated via the dynamic variable `database` or the buffer-local variable `dbc-database`. For instance, auxiliary files often set the `print-name` slot of their associated databases.

Code in an auxiliary file should be specific to the particular database; more general code is best placed in a separate file which is `load`ed (or, better, `require`d) by the auxiliary file. For instance, if you want to permit EDB to manipulate files of type Foo, you should put all Emacs Lisp code that applies to every Foo file in one file ('`db-foo.el`', say), and then put `(require 'db-foo)` in the auxiliary file associated with a particular Foo file. (Alternately, you may autoload a function that will be called in the auxiliary file; function `db-tagged-setup` is autoloaded from '`db-tagged.el`' in this manner.) Either technique keeps auxiliary files simple and small and makes Foo-specific code

easier to debug, byte-compile, and load only once per session. These advantages easily outweigh the introduction of an extra file.

Since the format file has not yet been interpreted, the auxiliary file could even change the contents of the buffer (and so the apparent contents of the format file); such extreme trickiness is only called for in special circumstances.

The format file can contain Emacs Lisp code in its "Local Variables" section; that code can do anything that code in the auxiliary file can do. If the format file is not named explicitly in the database (in the `default-format-file` slot), then function `db-file->format-file` tries to find one based on the database file name and the suffixes in `db-format-file-suffixes`; if that doesn't work either, you are prompted for a display format to use.

`db-format-file-suffixes`
: List of format file suffixes; the basename is that of the database file. The suffixes are tried in order; the default is (".dbf" ".fmt" "f"). The . that may precede the extension must be specified explicitly.

`db-format-file-path`
: List of directories (strings) to search, in order, for format files not found in the directory with their associated databases.

Code in the format file is useful for customizations specific to a particular format (such as setting variables which are local to the data display buffer); they can also be used for database-specific customizations if the database designer is sure that the file will always be the primary (first-selected) format for the database.

15.2 Hooks and customization functions

Hooks are variables whose values are "hook functions" (or lists of hook functions) which are called at particular times, such as when EDB has finished loading (to permit you to load customization code) or when a value has just been changed. Since hook functions can contain arbitrary code, they permit very powerful customizations. Customization functions are similar, but are single functions, never lists. Sometimes "hook" is used to mean either a hook or a customization function.

The following sections describe EDB's hooks and customization functions. Many of these hooks are change hooks, which permit a function (or functions) to be run whenever a value changes. These change hooks may be divided into two basic types: format change hooks and recordfieldspec

change hooks. The former are associated with a particular display format and are invoked when the value in a particular field, or in any field, changes. The latter (which are not yet implemented) are associated with a recordfieldspec and are invoked whenever a database record slot of a particular type is changed.

15.2.1 Load and read hooks

After EDB has finished loading, the following hook is run. This permits user customizations to be loaded automatically when EDB is (rather than being loaded unconditionally in the '`.emacs`' file, for instance), and permits users to change the definitions of functions defined by EDB, if desired.

`db-load-hooks`
: Function or list of functions run after loading EDB. You can use this to load extensions, redefine EDB functions, customize key bindings, etc.

The following two hooks are useful for causing database values seen by EDB to be different than those in the database file. The first can be used to modify the database file before it is read in; the second can be used to modify the database after it has been read in but before the user has had a chance to see it.

`db-before-read-hooks`
: Function or list of functions run immediately before a database is first read but after all local variables are set. The hooks are run in the data display buffer with variable `database` bound. Variable `db-buffer` is bound to a buffer containing the database file.

  This is a global variable. If you set it to be specific to a particular database (for instance, in the format or auxiliary file), then consider having its last action be to reset the variable to `nil`.

`db-after-read-hooks`
: Function or list of functions run after a database is completely read. The hooks are run in the data display buffer with variable `database` bound. For databases with nonregular layouts, you might put a call to `database-stored->actual` here, for instance.

  This is a global variable. If you set it to be specific to a particular database (for instance, in the format or auxiliary file), then consider having its last action be to reset the variable to `nil`.

15.2.2 Database minor mode hooks

EDB provides hooks that are run whenever the data display buffer is switched between Database View mode and Database Edit mode and which are run when a summary buffer is created.

`db-view-mode-hooks`
: Function or list of functions called when Database View mode is entered.

`db-edit-mode-hooks`
: Function or list of functions called when Database Edit mode is entered.

`database-summary-mode-hooks`
: Function or list of functions run when switching to Database Summary mode.

15.2.3 Record display hooks

The following function is run by the `display-record` function, which places a record's values in a data display buffer, each time a record is about to be displayed.

`dbf-before-display-record-function`
: A function called before a record is displayed by `display-record`. The function takes one argument, the record.

  This is a good place to put calls to `db-change-format`. Depending on your function's implementation, however, you may silently override any user calls to that function.

Here is an example of how you might use this:

```
(defun set-format-from-data (record)
  (if (< 0 (record-field record 'net-profit dbc-database))
      (db-change-format "loss format" "~/acct/db/loss.fmt")
    (db-change-format "profit format" "~/acct/db/profit.fmt")))

(setq dbf-before-display-record-function 'set-format-from-data)
```

This uses two different display formats, depending on the value of one field of a record. As you move from record to record in the database, each one is shown using the appropriate display format. A preferable implementation hides the filenames from the calls to `db-change-format` and instead uses, in the format or auxiliary file,

```
(setq dbf-format-name-spec-alist
```

```
'(("loss format" . "~/acct/db/loss.fmt")
  ("profit format" . "~/acct/db/profit.fmt")))
```

See the example file '`arb-demo.dba`' for an example of this. `dbf-format-name-spec-alist` need not specify the full pathnames if the format files are located in the same directory as the database or if '`"~/acct/db"`' is placed in `db-format-file-path`.

It would also be profitable to set `dbf-format-name` to whichever the first format was (this could be done in the format file's "Local Variables" section) so that the first-selected buffer wouldn't get read in twice (once when the database was read in and once when `display-record` was first called).

Finally, the database designer would probably arrange for there to be a change hook on the net-profit field so that when its value changed, the record could be redisplayed in the appropriate format automatically.

Here is an example of how to change the data display format when the user changes a particular field.

```
(defun equip-dbf-from-field (fieldname ignore-oldval ignore-newval)
  (let ((dbf-elt (assoc newval dbf-format-name-spec-alist))
        (save-index dbf-this-field-index))
    (db-change-format (if dbf-elt newval "default"))
    (setq dbf-this-field-index save-index) ; otherwise aref on nil
    ;; tell caller to redisplay record
    t))

(setq db-after-read-hooks
  (function (lambda ()
    (dbf-set-change-function 'equip-type 'equip-dbf-from-field))))
```

15.2.4 Edit mode hooks

These hooks are called whenever you enter a field to edit it, which provides an easy way to customize the behavior of particular format fields.

`dbf-enter-field-hook`

A function (of no arguments) called whenever a display field is entered. The displayspec index is `dbf-this-field-index`.

It is sometimes advantageous to have a particular action happen only once per edit of a record. For instance, when a record's address, city, state, or zip-code fields are edited, we might like to copy all the values to the old-address, old-city, old-state, and old-zip-code fields. We only want this to happen once, however: if you edit first the address, then the city, we don't want to repeat the process, because then the old-address field would get written over by the new value of the address field.

One way to prevent this from happening more than once is to set a variable when the copying is done, and then don't do the copying if that variable is set. The variable would be reset whenever a new record was edited. The following variable, which contains a list of other variables to reset each time Database Edit mode is entered, can accomplish just what is desired, when combined with a judicious use of change hooks.

`dbf-reset-on-edit-list`
: An alist of (variable-name . default-value) pairs. Every time Database Edit mode is entered, these buffer-local variables are reset to their default values. This is good for making sure that something only happens once each time a record is edited.

Here is an example (taken from an auxiliary file) of how to use `dbf-reset-on-edit-list`:

```
(setq db-after-read-hooks
      (function (lambda ()
         (setq dbf-reset-on-edit-list
               '((tep-homephone-oldified . nil)))
         (dbf-set-change-function 'home-phone 'tep-homephone-change-hook)
         (setq db-after-read-hooks nil))))

(defvar tep-homephone-oldified nil)

(defun tep-homephone-change-hook (fieldname oldvalue newvalue)
  (if (not tep-homephone-oldified)
      (dbf-this-record-set-field 'old-home-phone oldvalue))
  ;; return value indicates whether whole record should be redisplayed
  (prog1
      (not tep-homephone-oldified)
    (setq tep-homephone-oldified t)))
```

15.2.5 Display format change hooks

The following hook is run whenever a new record is created.

`db-new-record-function`

Function called on empty records before they're inserted in the database. Takes two arguments, the record and the database.

A typical use is to set default information or add a timestamp. For instance:

```
(defun set-update-date (record database)
  "Provide defaults for new records in the database."
  (record-set-field record 'updatedate
                    (parse-date-string (current-date)) database))
(setq db-new-record-function 'set-update-date)
```

The display format change hooks are called when you change a record field value. There are separate change hooks that run the first time any field is modified, whenever any field is modified, and whenever a particular field is modified. They run in the order `dbf-first-change-function`, `dbf-every-change-function`, and finally one of the elements of `dbf-change-functions`. Each change hook is either `nil` or a function of three variables: the fieldname of the just-modified field (a symbol) and the pre- and post-modification field values. These functions can set variable `dbf-redisplay-entire-record-p` to a non-`nil` value in order to cause the entire record to be redisplayed (for instance, if if the change hook modifies fields other than that named by its first argument).

`dbf-first-change-function`

A function called the first time a record field is modified, or `nil`. The function takes the fieldname and the old and new values as arguments, and returns `t` if the record should be redisplayed.

Here is an example of code to update the last modification field of a record, assuming its type is date:

```
(defun update-last-modified-date (fieldname oldvalue newvalue)
  "Put the current date in this record's `last modified' field."
  (dbf-this-record-set-field 'last-modified
                             (parse-date-string (current-date))))

(setq dbf-first-change-function 'update-last-modified-date)
```

`dbf-every-change-function`

A function called whenever a record field is modified, or `nil`. The function takes the fieldname and the old and new values as arguments, and returns `t` if the record should be redisplayed.

`dbf-change-functions`

A vector of one function (or `nil`) per record field (not display field). The functions take the fieldname and the old and new values as arguments, and return `t` if the record should be redisplayed. Use `dbf-set-change-function` to set the fields of this vector.

`dbf-set-change-function`

Set the change function for *fieldname* to *function* in the current database. *function* takes the fieldname and the old and new values as arguments, and returns `t` if the record should be redisplayed.

It is easy to make a field's value dependent on that of another field. For instance, suppose a salesman's commission should be 10% of the selling price of an item, and both fields are of type number. The database designer might choose to make the commission field unreachable (see Section 14.5 [Display specification optional parameters], page 64) and compute it whenever the selling price field varies. The latter operation could be done as follows:

```
(defun set-commission-from-selling-price (fieldname oldvalue newvalue)
  (dbf-displayed-record-set-field 'commission (/ newvalue 10)))

(dbf-set-change-function 'selling-price 'set-commission-from-selling-price)
```

There are two things to notice about this example. First, we need not set `dbf-displayed-record-set-field`, as `dbf-redisplay-entire-record` does that automatically. Second, if we replaced `(/ newvalue 10)` by `(/ (dbf-displayed-record-field 'selling-price) 10)`, then the function would work even if not called as a change function for selling-price.

You may modify records explicitly by calling `dbf-displayed-record-set-field` (see Section 16.3 [Manipulating records], page 86); when that is done, the following hook is invoked. It is different from the above functions in that they are called when you edit a field, while it is called when Emacs Lisp code modifies a field (usually as a result of some user action).

`dbf-set-this-record-modified-function`

A function called when the current record is marked as modified. The function takes no arguments and its return value is ignored. It is called after `dbf-this-record-original` is copied to `dbf-this-record` and after `dbf-this-record-modified-p` is set to `t`.

Another function is invoked when changes to a record are committed—that is, when changes to the record which is being displayed are copied back into its original in the database.

`dbf-after-record-change-function`

Function called whenever changes to a record are recorded semi-permanently by `dbf-process-current-record-maybe`. For convenience, the function takes the record as an argument, which is guaranteed to be `dbf-this-record`. Its return value is ignored.

15.2.6 Recordfieldspec change hooks

Recordfieldspec change hooks are not yet implemented.

15.3 Local variables

Variables may be specified to be local to a particular data display buffer or to a database; that is, when the variable's value is changed in one data display buffer or in one database, its value elsewhere is unaffected.

15.3.1 Per-data-display-buffer variables

Per-data-display-buffer variables permit different data display buffers to have different values for variables. This feature is heavily used by the EDB implementation; for instance, the per-buffer variable `dbc-database` records which database the data display buffer is displaying. Per-data-display-buffer variables are also useful when several data display buffers are all displaying the same database. The built-in Emacs function `make-variable-buffer-local` makes an ordinary variable local to every buffer.

`db-default-field-type`
: The type to use for record fields whose type is not explicitly specified.

15.3.2 Per-database variables

Per-database variables permit every data display buffer viewing a particular database to share information without making it global or interfering with other databases and other data display buffers. When the database is saved in internal EDB file layout, per-database variables are also saved, so their values persist from one invocation of the database to the next. Use the following functions to create and manipulate per-database variables.

`database-make-local`
: Declare a database-local variable named by *symbol* for *database*. Each such variable should only be declared once. If optional argument *value* is specified, the variable is set to it.

Database designers who are very concerned about speed should arrange to call this function in increasing order of frequency of variable reference; that is, add the least-used variables first.

`database-local-p`
: Return non-`nil` if *symbol* is a database-local variable for *database*.

`database-set-local`
: Set the value of database-local variable *symbol*, in *database*, to *value*. *symbol* must have been declared by a previous call to `database-make-local` unless optional argument *no-error* is supplied, in which case the function does that automatically, if necessary.

`database-get-local`
: Return the value of database-local variable *symbol* for *database*. If *symbol* was not declared by a previous call to `database-make-local`, an error is signalled unless optional argument *no-error* is non-`nil`, in which case `nil` is returned.

To learn how to set local variables automatically whenever a record is edited, See Section 15.2.4 [Edit mode hooks], page 72.

(This section of the manual does not refer to the "Local Variables" section of the database format file, which may be used to set variables and execute arbitrary Emacs Lisp code when a data display buffer is being set up; that is described in Section 13.5 [Reading from disk], page 55.)

15.4 Global variables

This section describes a potpourri of customization variables which you can use to control EDB's behavior.

When a potentially slow computation is underway, EDB displays a message in the echo area reporting how many records have been processed. Use the following variable to control how often this message is updated.

`db-inform-interval`
: When doing a lengthy computation, inform the user of progress every this many records. If `nil`, don't inform.

To make EDB use `with-electric-help` where appropriate instead of `with-output-to-temp-buffer`, set the following variables, which default to `nil`. You must have '`ehelp.el`' on your load path or have already loaded it.

`use-electric-help-p`

Non-`nil` if Emacs programs should use electric help where possible. Don't set this to a non-`nil` value unless the ehelp package is available.

EDB does not simply test for `(featurep 'ehelp)` because some packages load '`ehelp`' without determining whether you desire that behavior. Even if that has happened, users of EDB have a way to retain Emacs's traditional behavior.

`with-electric-help-maybe`

Similar to `with-electric-help` if `use-electric-help-p` is non-`nil`; otherwise like `with-output-to-temp-buffer` with the "*Help*" buffer. Ehelp is loaded if necessary. *body* is not a thunk (a function of no arguments), as with `with-electric-help`, but simply a set of forms.

16 Database representation

Perhaps the most important information about a database—besides the records it contains—is the number of fields in each record, and the type of each field. As explained in Section 1.5 [Terminology], page 6, a database consists of records with identical numbers of fields; each field has an associated type such as string or integer. Each field also has a name which is used when extracting its value from the record.

A database is basically just a doubly-linked circular list, where each link contains a single record. The database contains some additional supporting information, and so does each link in the list of records. The database is represented as a structure of type database whose `first-link` slot points to the circular list of links. Internally, database records are represented by vectors; however, the programmer should never manipulate those vectors directly, only through the functions described in this chapter.

Given a record, it is not possible to determine which link (if any) points to it; similarly, you cannot go from a link to its containing database. The database-to-link and link-to-record connections are one-way.

16.1 The database structure

The internal representation of a database is as a structure of type database. The slots of this structure may be accessed by using the macro `database-`*slotname*. The slots may be set using the macro `database-set-`*slotname*, whose second argument is the value to be stored in the slot.

The slots of the database are as follows:

`print-name`
: A string which briefly describes the database. It appears, among other places, in prompts for questions regarding the database. It defaults to "Unnamed database *n*", where the positive integers are assigned to *n* in order.

`first-link`
: The first link in the database. The links are arranged as a doubly-linked circular list, and each link contains a record, among other information. See Section 16.1.1 [The link structure], page 84.

`no-of-records`
: An integer, the number of records (and links) in the database. The first link is numbered 1 and the last link is numbered `no-of-records`.

`file`
: A string, the name of the file from which this database was read.

`file-local-variables`
: A string, the text of the "Local Variables" section of the file from which this database was read, if any.

`aux-file`
: A string, the name of this database's auxiliary file. If it is `nil`, then a number of default filenames are tried, based on `db-aux-file-suffixes` (see Section 15.1 [Auxiliary files], page 68).

`data-display-buffers`
: A list of data display buffers which are displaying this database. Since every summary buffer is associated with (and subordinate to) a data display buffer, summary buffers are not listed in the database structure.

`default-format-file`
: A string, the name of the default format file for this database. If it is `nil`, then a number of default filenames are tried by function `db-file->format-file` (see Section 15.1 [Auxiliary files], page 68).

`hide-functions`
: This does not appear to be used at present.

`no-of-fields`
: An integer, the number of fields in each record.

`fieldnames`
: A vector of symbols, the names of the record fields. Function `fieldnumber->fieldname` (see Section 16.3.2 [Accessing record fields], page 87) uses this to determine the name of a field, given its index.

  You may set this slot to be a list, and EDB will automatically convert it to a vector, as well as setting other database slots that can be determined from it. (When doing so, use `database-set-fieldnames-to-list`, which can safely be placed in any format file, instead of `database-set-fieldnames`; for details, see Section 10.1 [Changing display formats], page 29.) This information is duplicated in the recordfieldspecs.

  `database-set-fieldnames-to-list`
  : Set *database*'s fieldnames and record field types according to *fieldnames-list*. Users should not call `database-set-fieldnames` directly. *fieldnames-list* is a list of fieldnames (symbols); each list element may instead be a cons of fieldname and type to specify the field's recordfieldtype as well. If no type is specified for a field, the value of `db-default-field-type` is used.

This function sets several database slots besides the fieldnames slot, but has no effect if the fieldnames slot of the database is already set.

The following call specifies three fields of types one-line-string, integer, and string, presuming that variable `db-default-field-type` (see Section 15.3.1 [Per-data-display-buffer variables], page 76) has not been changed from its default of `string`:

```
(database-set-fieldnames-to-list database '((name . one-line-string)
                                            (age . integer)
                                            address))
```

`fieldname-alist`

An alist of fieldnames and indices. Function `fieldname->fieldnumber` (see Section 16.3.2 [Accessing record fields], page 87) uses this to determine the index of a field, given its name.

`recordfieldspecs`

A vector of symbols or recordfieldspecs which specify the type of each record field. If the value is a symbol, it is the name of a record field type name. The name is converted to a recordfieldspec via function `recordfieldtype->recordfieldspec`, which performs a lookup in `db-recordfieldtypes`. To access or change a particular recordfieldspec, use the following functions:

`database-recordfieldspec`

Return the recordfieldspec of *database* corresponding to *record-index*. Dereferences via `recordfieldtype->recordfieldspec` any symbol found in the recordfieldspecs slot of *database*.

`database-recordfieldspec-type`

Return the type of the recordfieldspec of *database* corresponding to *record-index*.

`database-set-recordfieldspec`

Set the recordfieldspec of *database* corresponding to *record-index* to *rs*. Use this to redefine, on a per-field basis, subfields of the recordfieldspec.

`field-priorities`

Determines in which order fields are compared when sorting records, and which fields are ignored entirely.

This slot's value is a cons of two lists: the first list contains fields that are used for sorting, and the second list is the ignored fields. Each list consists of pairs of fieldnumber and order-info. You may use `nil` for the second list when setting this slot. EDB always maintains the list of ignored fields, however, as its order might be worthwhile—for instance, for reminding the user of what the order used to be.

The order-info specifies how the field should be sorted: in increasing order, in decreasing order, or according to an arbitrary function. To choose the default ordering, or its

inverse, use the symbol `increasing` or `decreasing`. Otherwise, order-info is a cons of *type* and *value*, where *type* is a symbol (either `order-function` or `sort-function`) and *value* specifies the function.

In the sorting section's example (see Chapter 6 [Sorting], page 19), the record fields were '`firstname`', '`lastname`', '`address`', '`city`', and '`zip`'. Records were to be sorted on '`lastname`' in increasing order, or on '`firstname`' in decreasing order if the '`lastname`' fields matched, ignoring the other fields entirely for the purposes of the sort. The corresponding field priorities list would be

```
(((1 . increasing) (0 . decreasing)) .
 ((2 . increasing) (3 . increasing) (4 . increasing)))
```

`hidden-to-end-p`

A boolean which determines whether, when sorting, hidden records should be sorted in the usual way or placed at the end of the sorted order.

`internal-file-layout-p`

A boolean which determines whether the database is saved in internal file layout. This has no effect when the database is read, but it is set at read time so that, by default, the database is written out as it was read in.

Setting this slot, then saving the database to disk, is a good way to convert the database to or from internal file layout. It can be set in the usual way, or interactively via use of the following function:

`M-x db-toggle-internal-file-layout`

Toggle whether the database will be saved in EDB's internal file layout. With a nonzero prefix argument, set it to use internal file layout. With a zero prefix argument, set it not to use internal file layout.

`record-sepinfo`
`field-sepinfo`
`alternative-sepinfo`

These sepinfos are used when reading databases with delimited file layouts. A sepinfo contains a particular string, a regular expression, or a function that specifies how pieces of information are separated in the disk file (for more about the sepinfo structure, see Section 13.2.1 [How to specify delimited file layouts], page 45). These sepinfos describe how to separate records, fields within a record, and alternatives within a field. (The latter is not yet fully implemented.)

`read-record-from-region`

`nil` or a function of no arguments which returns a record read from the current region of the current buffer. For details, see Section 13.4 [Nonregular file layout], page 52.

`write-region-from-record`
: `nil` or a function which takes a record as its argument and inserts the file representation of that record in the current buffer. For details, see Section 13.4 [Nonregular file layout], page 52.

`sub-fieldsep-string`
`sub-recordsep-string`
: When delimiter substitution is required in reading a database, these strings are temporarily used to delimit fields and records, respectively. (These strings replace the actual field and/or record separators before substitution occurs.) Their values are chosen automatically if these slots aren't set.

`substitutions`
: An alist of actual and stored strings which permits translations from how the data appears in the data file to how it should really look; for instance, in data files with the tab-separated text layout, fields may not contain newlines, so any newlines in the data can be converted to some other character (such as '`^K`') when the database is written and then converted back when it is read in again. For more about substitution in reading databases, see Section 13.2 [Delimited file layout], page 44.

`modified-p`
: Non-`nil` if this database has been modified since it was last read or written.

`modifiable-p`
: Non-`nil` if this database may be modified. It is set to `nil` if the database file is not writable, and occasionally for other reasons. This does not prevent you from entering Database Edit mode, only from making changes while in Database Edit mode.

  The slot may be set directly. The following function, which is bound in Database View, Database Edit, and Database Summary modes, permits the slot to be changed interactively.

  `C-x C-q` Toggle whether the database may be modified by the user (`db-toggle-modifiable-p`). With a nonzero prefix argument, set it modifiable. With a zero prefix argument, set it non-modifiable.

`locals` An association list of symbols and values for per-database variables. (For the number of local variables I expect databases to have, an alist is faster than a hashtable, and it's easier to save to disk besides.) Such variables should be created with `database-make-local`, set using `database-set-local` (note the singular form) and dereferenced with `database-get-local`; for more information about these functions, see Section 15.3 [Local variables], page 76.

16.1.1 The link structure

The records of the database—the information that the user cares most about—are kept in a doubly-linked list, one record per link. The link structure also contains some other information about the record which doesn't belong in the record proper. The slots of a link are listed below; a slot may be accessed by using the macro `link-`*slotname* and set using the macro `link-set-`*slotname*, whose second argument is the value to be stored in the slot.

`prev` The previous link in the circular list.

`next` The next link in the circular list.

`hiddenp`
`markedp` These booleans are non-`nil` if this record is marked or hidden, respectively. For more information about marking and hiding, see Chapter 8 [Marking and hiding], page 25.

`summary` A string which is used to represent this record in the summary buffer, or `nil` if the record's value has changed since the last summary buffer was made (or if no summary buffer has been made).

`record` The database record proper, a vector with as many elements as the record has fields. Setting this slot with the `link-set-record` function also has the effect of setting the `summary` slot to `nil`, which is usually what is desired; to set only the `record` slot, use the `link-set-record-slot` macro instead.

16.2 Mapping over the database

Mapping refers to applying a function to each link or record in the database, or executing a piece of code for each link or record. Four functions provided this capability. The first two, more complicated, ones, provide access to each link of the database in turn.

`maplinks` Apply *maplinks-func* to every link in *database*. If optional third arg *hide* is non-`nil`, apply *maplinks-func* only to unhidden links. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` links. If optional fifth arg *accumulate* is non-`nil`, return a list of the results; otherwise return `nil`.

In the body, variable `maplinks-index` is bound to the index of the link being operated upon, and `maplinks-link` is the argument to *maplinks-func*. The loop may be short-circuited (aborted) by calling `maplinks-break`. To avoid the per-link function call overhead, use `maplinks-macro` instead.

`maplinks-macro`

Execute *maplinks-body* for each link in *database*, and return `nil`. If optional third arg *hide* is non-`nil`, execute *maplinks-body* only for unhidden links. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` links.

In the body, variable `maplinks-link` is bound to the link being operated upon, and `maplinks-index` is bound to its index. The loop may be short-circuited (aborted) by calling `maplinks-break`. Speed demons should call this instead of `maplinks` to avoid a function call overhead per link.

`maplinks-break`

Cause the maplinks loop to quit after executing the current iteration. This is not a nonlocal exit! It sets a flag which prevents future iterations. (Actually, it sets variable `maplinks-link`.)

Two other functions provide a slightly different interface which simplifies access to each record. Links and the information contained in them are not accessible from database records, but when that information is not of interest, these functions provide direct access to records.

`maprecords`

Apply *maprecords-func* to every record in *database*. Return `nil`. If optional third arg *hide* is non-`nil`, apply *maprecords-func* only to unhidden records. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` records. If optional fifth arg *accumulate* is non-`nil`, return a list of the results; otherwise return `nil`.

This is syntactic sugar for a call to `maplinks`, which see. See also `maprecords-macro`.

`maprecords-macro`

Execute *maprecords-body* for each record in *database*, and return `nil`. If optional third arg *hide* is non-`nil`, execute *maprecords-body* only for unhidden records. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` links.

In the body, variable maprecords-record is bound to the record being operated upon. The loop may be short-circuited (aborted) by calling `maprecords-break`.

This is syntactic sugar for a call to `maplinks-macro`, which see. See also `maprecords`.

`maprecords-break`

Cause the maplinks loop to quit after executing the current iteration. This is not a nonlocal exit! It sets a flag which prevents future iterations. (Actually, it sets variable `maplinks-link`.)

For instance, to sum, for all records, the values contained in field '`summand`' (of type number), you could use any of the following forms, presuming that variable `database` was set to the database in question:

```
(let ((result 0))
  (maplinks-macro
    (setq result (+ result (record-field (link-record maplinks-link)
                                         'summand database)))
    database)
  result)

(let ((result 0))
  (maprecords-macro
    (setq result (+ result (record-field maprecords-record
                                         'summand database)))
    database)
  result)

(let ((result 0))
  (maprecords
    (function (lambda (record)
      (setq result (+ result (record-field record 'summand database)))))
    database)
  result)

(apply (function +)
       (maprecords
         (function (lambda (record) (record-field record 'summand database)))
         database nil nil t))
```

16.3 Manipulating records

A database consists of records, each of which has the same makeup: corresponding fields in a database's records contain data of the same type. For instance, the fifth field of each record might contain an address, and the seventh field, a date. The particular addresses and dates would would vary from record to record. (Different databases will contain records with different numbers and types of fields.) Each field has a name and a type, which specifies what sort of information can be stored in the field; for more details about record field types, see Chapter 12 [Record field types], page 37.

Records are represented internally as vectors, but should never be operated on as such; use the abstractions described in this section.

16.3.1 Creating and copying records

`make-record`
: Return a record with number of fields specified by argument *database*.

  When you create a new record by using `db-add-record` (see Section 3.5 [Adding and removing records], page 11), `db-new-record-function` is invoked (see Section 15.2.5 [Display format change hooks], page 73), the number of records in the database is modified, and so forth. `make-record`, on the other hand, performs none of these housekeeping tasks.

`copy-record`
: Return a copy of *record*.

`copy-record-to-record`
: Copy the field values of the *source* record to the *target* record.

16.3.2 Accessing record fields

Ordinarily, record fields are accessed by specifying the name of the desired field; the database must also be specified so that the fieldname-to-fieldnumber correspondence can be determined.

`record-field`
: Return from *record* the field with name *fieldname*. Third argument is *database*.

`record-set-field`
: Set, in *record*, field *fieldname* to *value*. Fourth argument is *database*. Check constraints first unless optional fifth argument *nocheck* is non-`nil`.

There are also special commands for manipulating the current record—that is, the one that appears in the data display buffer. These functions require fewer arguments, flag that a redisplay of the record is necessary, and automatically call `dbf-set-this-record-modified-p`, which is essential if the changes are to be copied back into the original record in the database from the one that is being displayed. (A copy is always displayed so that changes can be undone.)

`dbf-displayed-record`
: Return the record currently displayed in this data display buffer. This is `dbf-this-record` if `dbf-this-record-modified-p` is non-`nil` and `dbf-this-record-original` otherwise.

`dbf-displayed-record-field`
: Return the value of the field named *fieldname* from the displayed record.

`dbf-displayed-record-set-field`
: Set field with name *fieldname* in displayed record to *value*. Cause the entire record to be redisplayed soon.

`dbf-displayed-record-set-field-and-redisplay`
: Set field with name *fieldname* in displayed record to *value*. Cause the entire record to be redisplayed immediately.

`dbf-set-this-record-modified-p`
: Set the value of `dbf-this-record-modified-p` to *arg*. If *arg* is non-`nil` and `dbf-this-record-modified-p` is `nil`, also do the necessary record-copying and call `dbf-set-this-record-modified-function`.

It is also possible—and more efficient—to use the fieldnumbers directly. The database does this internally, remembering fields by their numbers and only converting to fieldnames when interacting with the user. Adopting such a strategy for all field accesses would be cumbersome, error-prone, and make reading code difficult, but in some situations—particularly when `record-field` or `record-set-field` is being called with a constant second argument—it is worthwhile. The code can be sped up by allocating a variable for the fieldnumber, looking it up after the database has been loaded (for instance, by calling `fieldname->fieldnumber` after `database-set-fieldnames-to-list` or in `db-after-read-hooks`), and then using that variable along with `record-field-from-index` or `record-set-field-from-index`.

Do not confuse the record fieldnumber, which describes in what order fields happen to occur in the database's internal representation of a record, with the format fieldnumber, which describes in what order fields are displayed in the data display buffer.

`fieldname->fieldnumber`
: Given a *fieldname* and *database*, return a record fieldnumber. Do not be fooled into thinking this is a format fieldnumber.

`fieldnumber->fieldname`
: Given a record *fieldnumber* and *database*, return a record fieldname. If *database* is not specified, the value of `dbc-database` is used. The first argument is not a format fieldnumber.

`record-field-from-index`
: Return from *record* the value of the *fieldno*th field.

`record-set-field-from-index`
: Set, in *record*, the *fieldno*th field to *value*. Checks field constraints first if *database* is non-`nil`.

16.3.3 Mapping over record fields

To perform an action on every field of a record, use the following function or macro.

`mapfields`
: Apply *func* to each field in *record*, with variable `mapfields-index` bound. Third argument is *database*.

`mapfields-macro`
: Execute *body* for each field of *record*, a record of *database*, with variables `mapfields-field` and `mapfields-index` bound.

17 Naming conventions

17.1 Function and variable naming conventions

The names of EDB's functions and variables contain one of the following prefixes:

`edb-` These variables contain information about EDB such as the version number, last modification date, or names of the files comprising EDB. They do not relate to general database functionality, only to this particular implementation.

`db-` In a variable, indicates that the variable is global and affects all databases. In a function, indicates that the function is user-visible and may be called interactively. It is also used in some situations for internal database functionality which is not connected with any particular buffer.

`database-`
These functions operate on (the internal representation of) the database structure itself.

`dbc-` Indicates a variable local to the data display buffer which refers to the current database (the database being manipulated by that data display buffer), or a non-user-visible function which manipulates such variables. The 'c' stands for "current."

`dbf-` Indicates a variable local to the data display buffer which controls some aspect of formatting, or a non-user-visible function which manipulates such variables. The 'f' stands for "format"; many such variables are intimately related to the format, and the data display buffer used to be called the format buffer.

`dbs-` Indicates a variable local to the summary buffer, or a summary buffer function. Since the summary buffer may disappear at any time, the summary buffer gets most of its information from the associated data display buffer's local variables.

`dbfs-` Indicates a variable which is too important to be kept only in the summary buffer (which may disappear at any time), but is so often used by the summary buffer that it would be inefficient to keep it only in the data display buffer. Such variables are kept in both the data display and summary buffers.

`dbsi-` Indicates a variable local to a sort interface buffer, or a sort interface function.

17.2 File naming conventions

By convention, database file names contain one of the following suffixes:

`.dat` These are database files proper; they contain the information that makes up the fields and records of the database. Database filenames may also contain no extension at all.

`.fmt` Format files control the structure of the data display buffer, which displays one record at a time.

`.dba` Auxiliary files contain arbitrary Emacs Lisp code; they can be used to define functions, set variables, or operate directly on the database.

Other suffixes—or none at all—can be easily used; for instance, see variables `db-format-file-suffixes` and `db-aux-file-suffixes` (see Section 15.1 [Auxiliary files], page 68).

For more information, see Section 1.3 [Invoking EDB], page 3.

18 EDB support

EDB is not supported.

I wrote EDB because the existing tools for manipulating structured information in Emacs were lacking features I considered important. EDB still contains some lacunae. Feel free to mention which ones are most important to you, so that the lengthy EDB "to do" list can be properly ordered—but there is no guarantee that any of those project will ever be undertaken. Even better, contribute code for features currently lacking—then you'll be sure of its inclusion in EDB, and you'll be helping others as well. Also send bug reports and (if possible) bug fixes for inclusion in future releases.

The mailing list '`edb-list@theory.lcs.mit.edu`' is intended for discussions relating to EDB: trading extensions, sharing experiences, asking questions, reporting bugs and bug fixes, and distributing updates to EDB. Send requests to be added to (or removed from) the list to '`edb-list-request@theory.lcs.mit.edu`'. The list is not archived; rather, the manual and examples distribution are updated with useful information that appears on the list.

The mailing list '`edb-developers@theory.lcs.mit.edu`' is intended for lengthier technical discussions about to EDB. '`edb-list`' is the mailing list of first recourse for EDB users; when discussions become too detailed to be of interest to most members, the conversations are moved to '`edb-developers`'. Discussions of tentative future plans and of bug fixes appear on this list; its members should have a good understanding of EDB. Send requests to be added to (or removed from) the list to '`edb-developers-request@theory.lcs.mit.edu`'. The list is not archived.

18.1 Updates to EDB

Periodically updates to EDB are sent to the mailing list '`edb-list@theory.lcs.mit.edu`'. You can make it easy to install these updates by putting something like the following in your '`.emacs`' file:

```
(setq edb-directory "~/emacs/edb")
(autoload 'edb-update "database" "EDB database package" t)
```

Then, when you receive a message containing diffs for a new version of EDB, you only have to type `M-x edb-update RET` in order to install them.

`M-x edb-update`

Install the EDB update found in the current buffer after point. EDB is assumed to be in the directory specified by `edb-directory`. (If that variable is not set, the user is prompted for the location of the files.)

If you have trouble with this command, it is likely that your version of EDB is not exactly the same as the last release. You might have an old release, or you might have a pre-release.

`edb-directory`

A string, the name of the directory containing the EDB source files.

18.2 In case of trouble

18.2.1 Variables

In some cases the documentation strings and/or default values of some variables may be missing—as if the variables were not yet defined. That's because the variables are not yet defined; they are associated with part of EDB which hasn't been loaded because it hasn't been needed yet. The documentation and default values will appear when that part of EDB is loaded (if you set such variables, your values will not be replaced). Such variables are correctly declared buffer-local (if appropriate), so you can set them without fear of the changes affecting other buffers.

18.2.2 Exiting Emacs or saving files

If you are unable to exit Emacs or to execute `db-save-some-buffers` (which replaced `save-some-buffers`, which is ordinarily bound to `C-x s`) because Emacs is trying to manipulate a database which doesn't exist or because an EDB bug is triggered by the attempt to save an existing database, you can set the variable `db-databases` to `nil`. This indicates to EDB that there are no databases read into memory and, therefore, no operations will be attempted on them as a part of saving all modified Emacs buffers.

`db-databases`

Assoc list of database names and databases.

18.2.3 Compiling EDB

It is strongly recommended that you run EDB byte-compiled, as otherwise it is very sluggish. One easy way to byte-compile EDB is to run '`make`' from the shell in the EDB directory. To byte-compile EDB from within Emacs, type `M-x byte-compile-database RET`. (You may need to load EDB first, by typing `M-x load-library RET database RET` or `M-x load-file RET database.el RET`, in order to define this function.) If you perform the byte-compilation yourself rather than using the `byte-compile-database` function, you **must** fully load the code before compiling it, by typing `C-u M-x load-database RET`.

Compilation of EDB may signal several warnings that the following functions are not known to be defined: `delete-extent`, `find-face`, `make-extent`, `map-extents`, `mouse-track`, `set-extent-face`, and `x-flush-mouse-queue`. You may ignore these warnings, which result from EDB support for multiple Emacs installations with slightly different feature sets and function names.

18.2.3.1 Load EDB before compiling it

You must always fully load EDB before attempting to byte-compile it. The easiest way to ensure this is to compile EDB by using function `byte-compile-database`, which automatically fully loads EDB.

Otherwise, do `C-u M-x load-database RET` to load a full uncompiled version of EDB before you compile. You may need to load EDB before doing this (in order to define the `load-database` function), but do not omit this step even if you have already loaded EDB. It is not enough to simply do '`(require 'database)`' or call `db-find-file`, since neither of those actions loads all of EDB, only parts of it.

There are two reasons for loading the code before compiling it. The first is that, when the byte-compiler encounters an unknown symbol used in function position, it assumes that it is a function and attempts to funcall it. If the symbol is later defined to be a macro, this leads to a runtime error. The second reason is that, if the variable `db-disable-debugging-support` is non-`nil` (most users will want to use the default value, which is `t`) when compilation occurs, then code for assisting debugging will be compiled out and the code will be slightly smaller and faster, because it will not contain conditional code for printing status reports and intermediate results.

One symptom of compiling EDB without having loaded it is a message along the lines of '`Invalid function: (macro ...)`'. (EDB's implementation uses macros for efficiency, so problems result if, when compiling, an unknown symbol is assumed to be a function but is actually later defined as

a macro.) Another symptom is that variables defined in autoloaded files will be reported as "not known to be defined." A problem with compiling EDB when an old version is loaded is that, if macro definitions have changed, the old definitions will be compiled into the new code.

`M-x load-database`
: Load all the files of EDB, the Emacs database. With prefix argument, load source, not compiled, code; run EDB interpreted.

`M-x byte-compile-database`
: Compile source (.el) files in EDB, the Emacs database, which need it. If optional prefix argument *all* is non-`nil`, every source file is recompiled. You are likely to want unconditional recompilation if you have tried to compile any of the files by hand and want to replace the (possibly incorrect) results.

`M-x byte-compile-database-all`
: Compile all source (.el) files in EDB, the Emacs database, unconditionally. Calls `byte-compile-database`.

18.2.4 Debugging EDB

18.2.4.1 Enabling debugging messages

Two useful sources of information for locating a problem in EDB are backtraces and the database log. If you encounter an EDB error, you should generate a backtrace and a database log; even if they do not help you, they may provide information to others who will see your bug report. Execute the following command and then repeat the commands that caused an error.

`M-x db-prepare-to-debug`
: Prepare to debug EDB. Set variables `debug-on-error`, `db-disable-debugging-support`, and `db-debug-p`. Also load uncompiled EDB source.

EDB's source code contains calls to debugging macros which print useful messages and save them in the '`*Database-Log*`' buffer for later examination. By default, the calls to these macros are removed at compile time; this results in slightly smaller, faster code. The following variables control this behavior.

`db-disable-debugging-support`
: If non-`nil`, then debugging calls will be compiled out of the source and the variable `db-debug-p` will have no effect. Setting this variable at run-time has no effect if you are running EDB compiled; you must set it when you compile EDB, or run EDB interpreted. Defaults to `t`.

`db-debug-p`
: Non-`nil` if database debugging is enabled. Defaults to `nil`. Has no effect on code compiled with `db-disable-debugging-support` set.

18.2.4.2 Printing circular structures

EDB's internal representation of the database structure is circular; if you try to print it, Emacs will signal an error, possibly after running for a very long time and appearing to hang. This is a particular problem when debugging functions that manipulate a database.

Emacs's custom-print package defines versions of the printing commands which support printing circular structures. (The Edebug debugger, which is worth using in its own right, supports custom-print's redefinitions.) To install custom-print, place the following in your '`.emacs`' file:

```
(require 'cust-print)
(custom-print-install)
```

Then, do something like After installing these packages (see their documentation for details), you can simply do something like

```
(setq print-level 4)
```

to make circular structures easily debugable.

Be sure to reset `print-level` and `print-length` to `nil` before byte-compiling! (If you do not, the byte-compiled code may contain a '`#`' or '`...`' in place of some of the byte codes, resulting in an error:

```
error  Invalid read syntax: "#"
```

One way to avoid the problem is to put the following code in your '`.emacs`' file:

```
(require 'advice)
```

```
(defadvice byte-compile-file (before undo-custom-print activate)
  (setq print-level nil
        print-length nil))
```

18.2.5 Reporting bugs

Report bugs and problems with EDB to '`edb-developers@theory.lcs.mit.edu`'; they don't promise to do anything but might well want to fix it. Questions about EDB that are not answered in the manual are welcome as well; if the manual is unclear, that's a bug in the documentation. Suggestions for new features or modifications are always welcome (and implementations of such features, even more so); input from users determines which features are moved to the front of EDB's "to-do" list and which ones users apparently don't care about. The '`edb-list`' mailing list (see Chapter 18 [EDB support], page 92) is also a valuable resource for users of EDB.

Before reporting a bug or trying to fix it yourself, please perform the following steps. First, make sure you are using the most recent version of EDB, since the problem may have already been fixed. The most recent version of EDB appears in directory '`theory.lcs.mit.edu:/pub/emacs/edb/`'; the file '`edb-`*version*`.tar.gz`' gives the version number. You can find out which version of EDB you have by looking in your version of '`database.el`' or by typing `C-h v edb-version RET`.

Second, read the appropriate manual sections, so you understand how EDB ought to behave (and whether your bug is considered a feature).

Third, try to isolate the problem (for instance, by using the smallest possible data, format, and auxiliary files). If the problem causes an error (usually causing Emacs to beep and display a message in the echo area), then run command `db-prepare-to-debug` and then reproduce the error. This will produce a backtrace and a database log. For further debugging techniques, see Section 18.2.4 [Debugging EDB], page 95.

Send a bug report which includes all of the following information.

- A description of how to reproduce the bug and what you expected to occur.
- The version of EDB you are running (do `M-x edb-version RET`) and the version of Emacs (do `M-x emacs-version RET`).
- **All files necessary to reproduce the bug**, including the database file, the format file, and the auxiliary file, if any. I promise not to look at the content of your database file or distribute it; if you feel uncomfortable sending it nonetheless, construct another one that produces the

same error. Reproducing these files usually takes longer than tracking down the bug; if they are absent, no action will be taken.

- The backtrace and the database log (found in the '`*Backtrace*`' and '`*Database Log*`' buffers).

Function Index

B

byte-compile-database 94, 95
byte-compile-database-all 95

C

copy-record 87
copy-record-to-record 87
copy-recordfieldspec 38

D

database-get-local 77
database-local-p 77
database-make-local 76, 77
database-recordfieldspec 81
database-recordfieldspec-type 81
database-set-fieldnames 80
database-set-fieldnames-to-list 34, 38, 50, 80
database-set-local 77
database-set-recordfieldspec 81
database-sort 41
database-stored->actual 70
date->storage-string 39
date->storage-string-lisp 39
date->storage-string-mmddyyyy 39
date-day 61
date-month 61
date-year 61
db-accept-record 11
db-add-record 11, 87
db-additional-data-display-buffer 31
db-change-format 29, 30, 71
db-commit-record 11
db-copy-record 11
db-delete-record 11
db-exit 12
db-field-help 15, 41
db-file->format-file 69
db-find-file 3
db-first-field 10, 14
db-first-record 9
db-hide-record 26
db-hide-unmarked-records 26
db-hiding-set 27
db-hiding-toggle 27
db-jump-to-record 9
db-kill-buffers 12
db-last-field 10, 14
db-last-record 9
db-mark-record 25
db-mark-unhidden-records 26
db-next-field 14
db-next-line-or-field 15
db-next-marked-record 27
db-next-record 9, 14
db-next-record-ignore-hiding 26
db-next-screen-or-record 9
db-output-record-to-db 11
db-prepare-to-debug 95
db-previous-field 14
db-previous-line-or-field 15
db-previous-marked-record 27
db-previous-record 9, 14
db-previous-record-ignore-hiding 27
db-previous-screen-or-record 9
db-quit 11
db-report 28
db-revert-database 10
db-revert-field 14
db-revert-record 10
db-save-some-buffers 93
db-search 16
db-search-field 16
db-setup-data-display-buffer 56
db-setup-format-parse-displayspecs 56
db-sort 19
db-summary 23
db-tagged-setup 50
db-this-buffer 4

db-toggle-internal-file-layout 82
db-toggle-modifiable-p 83
db-toggle-show-hidden-records 27
db-unhide-all 26
db-unmark-all 26
db-view-mode 13
dbc-set-hide-p 27
dbf-always 30
dbf-displayed-record 87
dbf-displayed-record-field 88
dbf-displayed-record-set-field 88
dbf-displayed-record-set-field-and-redisplay 88
dbf-process-current-record-maybe 75
dbf-set-change-function 74, 75
dbf-set-summary-format 23, 24, 31
dbf-set-this-record-modified-p 88
dbsi-decreasing 20
dbsi-increasing 20
dbsi-kill-line 19
dbsi-ordering-function 20
dbsi-quit 21
dbsi-quit-clear-buffer-default 21
dbsi-sorting-function 20
dbsi-this-field-only 21
dbsi-toggle-hidden-to-end 19
dbsi-use-ordering 21
dbsi-use-ordering-make-buffer-default 21
dbsi-use-ordering-make-database-default 20
dbsi-yank-line 19
debug-on-error 95
define-displaytype-from-displayspec 64
define-displaytype-from-optstring 63
define-enum-type 62
define-recordfieldtype-from-recordfieldspec .. 38
define-type-alias 40
delete-extent 94
display-record 31, 71
displaytype->displayspec 64

E

edb-after-find-file 4
edb-update 93

F

fieldname->fieldnumber 81, 88
fieldnumber->fieldname 80, 88
find-face 94
find-file 4
format-date 60
format-date-full 39

K

kill-buffer 12

L

link-set-record 84
link-set-record-slot 84
load-database 94, 95

M

make-date 61
make-displayspec 64
make-extent 94
make-n-line-sep-function 48
make-record 52, 87
map-extents 94
mapfields 89
mapfields-macro 89
maplinks 84, 85
maplinks-break 84, 85
maplinks-macro 84, 85
maprecords 85
maprecords-break 85
maprecords-macro 85
mouse-track 94

P

parse-date-string 39, 60

R

record-field 87
record-field-from-index 88
record-set-field 52, 87
record-set-field-from-index 89
recordfieldtype->recordfieldspec 38, 81
right-justify 65

right-justify display specification parameter 65

S

save-some-buffers 8, 93
set-extent-face 94
simple-format-date 40
storage-string->date 39, 40
storage-string-lisp->date 39
storage-string-mmddyyyy->date 39

W

with-electric-help 77, 78
with-electric-help-maybe 78
with-output-to-temp-buffer 77, 78

X

x-flush-mouse-queue 94

Variable Index

D

database ... 70
database-summary-mode-hooks ... 71
db-after-read-hooks ... 56, 70, 88
db-aux-file-path ... 68
db-aux-file-suffixes ... 68
db-before-read-hooks ... 56, 70
db-buffer ... 70
db-databases ... 93
db-debug-p ... 95, 96
db-default-field-type ... 76, 80, 81
db-delete-record-modifies-database-p ... 11
db-disable-debugging-support ... 94, 95, 96
db-edit-mode-hooks ... 71
db-format-file-path ... 69, 72
db-format-file-suffixes ... 69
db-inform-interval ... 77, 84, 85
db-load-hooks ... 70
db-new-record-function ... 74, 87
db-recordfieldtypes ... 81
db-sort-modifies-p ... 21
db-tagged-continuation ... 51
db-tagged-continuation-output ... 51
db-tagged-continuation-regexp ... 51
db-tagged-rrfr-hooks ... 51
db-tagged-separator ... 51
db-tagged-separator-output ... 51
db-tagged-separator-regexp ... 51
db-tagged-tag-chars ... 51
db-tagged-wrfr-after-hooks ... 51
db-tagged-wrfr-before-hooks ... 51
db-view-mode-hooks ... 71
dbc-database ... 88
dbc-hide-p ... 26, 27
dbf-after-record-change-function ... 75
dbf-always-forms ... 30
dbf-before-display-record-function ... 29, 30, 71
dbf-change-functions ... 74
dbf-enter-field-hook ... 72
dbf-every-change-function ... 74
dbf-field-priorities ... 20, 41
dbf-first-change-function ... 74
dbf-format-file ... 30
dbf-format-name ... 30, 71
dbf-format-name-spec-alist ... 29, 72
dbf-hidden-to-end-p ... 20
dbf-redisplay-entire-record-p ... 74
dbf-reset-on-edit-list ... 73
dbf-set-this-record-modified-function ... 75, 88
dbf-summary-function ... 23
dbf-summary-show-hidden-records-p ... 23
dbf-this-field-index ... 72
dbf-this-record ... 75, 87
dbf-this-record-modified-p ... 75, 87, 88
dbf-this-record-original ... 75, 87
dbm-string-prefix-regexp ... 66
dbsi-hidden-to-end-p ... 19
debug-on-error ... 97

E

edb-directory ... 92, 93

F

find-file-hooks ... 4

I

inhibit-local-variables ... 56

M

mapfields-field ... 89
mapfields-index ... 89
maplinks-index ... 84, 85
maplinks-link ... 84, 85

P

print-length ... 96
print-level ... 96

U

use-electric-help-p 78

Concept Index

*

'`*Database-Log*`' buffer 95

.

'`.dat`' file suffix 3
'`.dba`' file suffix 3
'`.emacs`' file 2, 92
'`.fmt`' file suffix 3

=

=, in search pattern 17

>

>, in search pattern 17

<

<, in search pattern 17

A

`a->d` display specification parameter 66
accepting changes 10
accessing record fields 87
`actual->display` displayspec slot 66
`actual->stored` recordfieldspec slot 41
adding a record 11
adding fields 35
additional data display buffers, making 31
after-find-file hooks 4
alternate display formats 29
`alternative-sepinfo` database slot 82
alternatives, in enumeration types 61
ambiguities in database files, resolving 48
apparently circular structure being printed 96
autoloading EDB 2
`aux-file` database slot 80
auxiliary file 68
auxiliary file name 68

B

backtrace, viewing circular structures 96
beginning of file, text at 45
boolean recordfieldspec 39
Botswana 54
bugs, reporting 97
byte-compiling EDB 94
byte-compiling EDB, trouble with 94

C

change hooks, for display formats 73
change hooks, for recordfieldspecs 76
`change-hook` recordfieldspec slot 42
changes, accepting them 10
changes, committing them 10
changes, making them permanent 10
changing display formats 29
circular structures, printing 96
committing changes 10
`common-form-function` recordfieldspec slot 41
compiling EDB 94
compiling EDB, trouble with 94
`constraint-function` recordfieldspec slot 42
converting a file to or from EDB internal layout 43
copying records 87
creating a database 33
creating records 87
custom-print package 96
customization 68
customization functions 69
customization, global variables 77

D

`d->a` display specification parameter 66
data display buffer, hiding fields 30
data display buffers, making additional 31
data file layout 43
data file layout, delimited 44
data file layout, internal 43

data file layout, nonregular ... 52
data file layout, tagged ... 50
data-dependent display format ... 71
data-display-buffer-local variables ... 76
`data-display-buffers` database slot ... 80
Database Edit mode ... 13
database file layout ... 43
database files, editing ... 7
database minor mode hooks ... 71
database representation ... 79
database structure ... 79
Database Summary mode ... 23
Database View mode ... 9
database-local variables ... 76
date displaytype ... 60, 61
date recordfieldspec ... 39
date-efficient-storage recordfieldspec ... 40
date-or-nil recordfieldspec ... 40
debugging EDB ... 95
debugging messages, enabling ... 95
`default-format-file` database slot ... 80
`default-value` recordfieldspec slot ... 41
defining displaytypes ... 63
deleting a record ... 11
deleting fields ... 35
delimited file layout ... 44
delimiters, record and field ... 44
dependent field values ... 75
designing a database ... 33
display format change hooks ... 73
display format file name ... 68
display format, alternate ... 29
display format, data-dependent ... 71
display format, selecting ... 29
display format, specifying ... 58
display format, variant ... 29
display specification ... 58
display specification optional parameters ... 64
`display->actual` displayspec slot ... 66
displayspec fields ... 64
displayspec structure ... 64
displaytype, compared to record field type ... 37
displaytype, defining ... 63
displaytype, not set by display specification ... 40, 59
displaytypes, predefined ... 60

E

EDB internal layout, converting to or from ... 43
EDB, new versions of ... 92
EDB, upgrades ... 92
'`edb-developers`' mailing list ... 92
'`edb-list`' mailing list ... 92
edit mode ... 13
edit mode hooks ... 72
editing database files ... 7
ehelp package ... 77
Einstein, Albert ... 54
electric help package ... 77
Emacs initialization file ... 2, 92
enabling debugging messages ... 95
end of file, text at ... 45
enforcing constraints ... 42
enumeration types ... 61
eval expressions, in format file ... 30
example databases, getting via ftp ... 2
exiting database mode ... 11
exiting Emacs, trouble with ... 93

F

field delimiters ... 44
field separator, setting ... 45
field type, record ... 37
`field-priorities` database slot ... 81
`field-sepinfo` database slot ... 82
`fieldname-alist` database slot ... 81
`fieldnames` database slot ... 80
fields, accessing them in records ... 87
fields, adding ... 35
fields, deleting ... 35
fields, reading them in records ... 87
fields, rearranging ... 35
fields, reordering ... 35
fields, setting them in records ... 87
fieldspec, see recordfieldspec ... 40
`file` database slot ... 80
file format for data file ... 43

file layout ... 43
file layout for data file ... 43
file layout, delimited ... 44
file layout, internal ... 43
file layout, nonregular ... 52
file layout, tagged ... 50
file name, for auxiliary file ... 68
file name, for display format ... 68
file naming conventions ... 90
`file-local-variables` database slot ... 80
files used by EDB ... 3
files, editing database ... 7
find-file-hooks ... 4
`first-link` database slot ... 79
floating-point number displaytype ... 60
floating-point number recordfieldspec ... 39
format file ... 68
format file name ... 69
format file, eval expressions in ... 30
format file, local variables section ... 30
format file, path to search ... 69
format file, primary ... 3, 30
format name ... 30
format, of data file ... 43
format-spec structure ... 29
function naming conventions ... 90

H

`height` display specification parameter ... 65
help for record fields ... 15
`help-info` recordfieldspec slot ... 41
`hidden-to-end-p` database slot ... 82
`hiddenp` link slot ... 84
`hide-functions` database slot ... 80
hiding ... 25
hiding fields from a data display buffer ... 30
hooks ... 69
hooks, change, for display formats ... 73
hooks, change, for recordfieldspecs ... 76
hooks, database minor mode ... 71
hooks, edit mode ... 72
hooks, record display ... 71

I

`indent` displayspec slot ... 65
inserting a record ... 11
installing EDB ... 2
integer displaytype ... 60
integer recordfieldspec ... 39
integer-or-nil displaytype ... 60
integer-or-nil recordfieldspec ... 39
internal data file layout ... 43
`internal-file-layout-p` database slot ... 53, 82
interpreted code, running ... 95
invalid function compilation error ... 94
invalid read syntax ... 96
invoking EDB ... 3

J

justification of display fields ... 65

K

killing a database buffer ... 11
killing a record ... 11

L

last modification field ... 74
layout, of data file ... 43
left justification of display fields ... 65
link structure ... 84
loading EDB ... 2
local variables ... 76
local variables section of format file ... 30, 54, 56, 80
local variables, per data display buffer ... 76
local variables, per database ... 76, 83
`locals` database slot ... 83
log, of debugging messages ... 95
looping over the database ... 84

M

mailing lists for EDB ... 92
making a database ... 33
making changes permanent ... 10
mapping over the database ... 84
`markedp` link slot ... 84
marking ... 25

`match-actual->display` displayspec slot 66
`match-display->actual` displayspec slot 66
`match-function` recordfieldspec slot 41
`max-height` displayspec slot 65
`max-width` displayspec slot 65
`merge-function` recordfieldspec slot 41
`min-height` displayspec slot 65
`min-width` displayspec slot 65
mode line 7
`modifiable-p` database slot 83
`modified-p` database slot 83
moving from field to field 14
moving from record to record 9, 14
multi-character enumeration displaytypes 62

N

name of a database 79
naming conventions for files 90
naming conventions for functions and variables 90
new records, setting default information 74
new versions of EDB 92
newline, at end of database file 49
`next` link slot 84
nil-or-string displaytype 60
nil-or-string recordfieldspec 39
`no-of-fields` database slot 80
`no-of-records` database slot 80
`noindent` display specification parameter 65
nonregular file layout 52
number displaytype 60
number recordfieldspec 39
number-or-nil displaytype 60
number-or-nil recordfieldspec 39

O

one-character enumeration displaytypes 62
one-line-string displaytype 60
one-line-string recordfieldspec 39
one-line-string-or-nil displaytype 60
one-line-string-or-nil recordfieldspec 39
`order-function` recordfieldspec slot 41
ordering functions 21

P

`padding-action` displayspec slot 65
per-data-display-buffer variables 76
per-database variables 76
`post-last-regexp` sepinfo slot 47
`post-last-regexp-submatch` sepinfo slot 47
`post-last-string` sepinfo slot 46
`pre-first-regexp` sepinfo slot 46
`pre-first-regexp-submatch` sepinfo slot 46
`pre-first-string` sepinfo slot 46
predefined displaytypes 60
predefined recordfieldspecs 39
`prev` link slot 84
primary format file 3, 30
`print-name` database slot 79
printing circular structures 96
problems, reporting 97

Q

quitting database mode 11

R

`reachablep` displayspec slot 67
`read-record-from-region` database slot 52, 82
reading a database from disk 3
reading from disk, details 55
reading large database is slow 43
reading record fields 87
rearranging fields 35
record delimiters 44
record display hooks 71
record field index 64
record field type 37
record field type, compared to displaytype 37
record field type, specifying 38
record fields, accessing 87
`record` link slot 84
record representation 86
record separator, setting 45
`record-index` displayspec slot 64
`record-sepinfo` database slot 82
recordfieldspec change hooks 76
recordfieldspec structure 40

`recordfieldspecs` database slot ... 81
recordfieldspecs, predefined ... 39
removing a record ... 11
reordering fields ... 35
reporting bugs in EDB ... 97
reporting problems with EDB ... 97
reporting trouble with EDB ... 97
reports ... 28
representation of database ... 79
resolving ambiguities in database files ... 48
reverting changes to a field ... 13
reverting changes to a record ... 10
right justification of display fields ... 65

S

saving files, trouble with ... 93
saving to disk ... 4
search patterns ... 16
searching ... 16
selecting only some records ... 25
`sep-function` sepinfo slot ... 46
`sep-regexp` sepinfo slot ... 46
`sep-regexp-submatch` sepinfo slot ... 46
`sep-string` sepinfo slot ... 46
separator, setting field ... 45
separator, setting record ... 45
sepinfo examples ... 47
sepinfo structure ... 45
setting record fields ... 87
simultaneously manipulating two records ... 31
simultaneously using two formats ... 31
slot assigners, for display specifications ... 64
slotsetters, for display specifications ... 64
slow reading of large databases ... 43
`sort-function` recordfieldspec slot ... 41
sorting ... 19
sorting functions ... 21
specifier, format ... 29
specifying a record field type ... 38
starting up EDB ... 3
`stored->actual` recordfieldspec slot ... 41
string displaytype ... 60
string recordfieldspec ... 39
string-or-nil displaytype ... 60
string-or-nil recordfieldspec ... 39
`sub-fieldsep-string` database slot ... 83
`sub-recordsep-string` database slot ... 83
substitution, in reading a database file ... 49
`substitutions` database slot ... 83
summary format, setting ... 23
`summary` link slot ... 84
summary mode ... 23

T

tab-separated text file layout ... 44
tagged file layout ... 50
texinfo, EDB uses version 2 ... 2
three file types used by EDB ... 3
time displaytype ... 61
time recordfieldspec ... 40
trouble with compiling EDB ... 94
trouble with exiting Emacs ... 93
trouble with saving files ... 93
trouble with undefined variables ... 93
trouble, reporting ... 97
Truman, Harry S ... 33
`trunc-display` display specification parameter ... 65
`trunc-edit` display specification parameter ... 67
`truncation-display-action` displayspec slot ... 65
`truncation-editing-action` displayspec slot ... 67
two formats, using simultaneously ... 31
two records, manipulating simultaneously ... 31
`type` recordfieldspec slot ... 40
type, display, defining ... 63
type, display, predefined ... 60
type, record field ... 37
type, specifying record field ... 38

U

undoing changes to a field ... 13
undoing changes to a record ... 10
uneditable fields in data display buffer ... 67
`unreachable` display specification parameter ... 67
upgrades to EDB ... 92

V

variable default value missing 93
variable documentation missing 93
variable naming conventions 90
variable not known to be defined 94
variables, per-data-display-buffer 76
variables, per-database 76
variant display formats 29
version number, of EDB 97
View mode, Database 9

W

`width` display specification parameter 65
`write-region-from-record` database slot 53, 83
writing to disk 4

Y

yes-no displaytype 60

Short Contents

1 Introduction 1
2 Database mode 7
3 Database View mode 9
4 Database Edit mode 13
5 Searching 16
6 Sorting 19
7 Database Summary mode 23
8 Marking and hiding 25
9 Reports 28
10 Specifying the display format 29
11 Designing a database 33
12 Record field types 36
13 Database file layout 42
14 How information is displayed 57
15 Customization 67
16 Database representation 78
17 Naming conventions 89
18 EDB support 91
Function Index 98
Variable Index 101
Concept Index 103

Table of Contents

1 Introduction 1

1.1 Organization of this manual 2
1.2 Installing EDB 2
1.3 Invoking EDB 3
1.4 Example EDB session 5
1.5 Terminology 6

2 Database mode 7

3 Database View mode 9

3.1 Moving around in the database 9
3.2 Changing to Database Edit mode 10
3.3 Undoing all changes to a record 10
3.4 Making changes permanent 10
3.5 Adding and removing records 11
3.6 Exiting database mode 11

4 Database Edit mode 13

4.1 Exiting Database Edit mode 13
4.2 Undoing changes to a field 13
4.3 Moving from record to record 14
4.4 Moving from field to field 14
4.5 Movement within a field 14
4.6 Editing a field 15
4.7 Getting help 15

5 Searching 16

5.1 Search patterns 16
5.1.1 Basic patterns 17
5.1.2 Comparisons 17
5.1.3 Logical connectives 17
5.1.4 Other pattern operations 18

6 Sorting 19

6.1 Sorting and ordering functions 21

7 Database Summary mode 23

8 Marking and hiding 25

8.1 Setting the mark and hide bits 25
8.2 Movement among marked and hidden records 26
8.3 Details of hiding 27

9 Reports 28

9.1 Bugs in report generation 28

10 Specifying the display format 29

10.1 Changing display formats 29
10.2 Execution of format file eval expressions 30
10.3 Making additional data display buffers 31

11 Designing a database 33

11.1 Creating a new database 33
11.2 Manipulating database fields 35

12 Record field types 36

12.1 Specifying a record field type 37
12.2 Predefined record field types 38
12.3 The recordfieldspec structure 39

13 Database file layout 42

13.1 Internal file layout 42
13.2 Delimited file layout 43
13.2.1 How to specify delimited file layouts 44
13.2.1.1 The sepinfo structure 44
13.2.1.2 Record and field separator examples 46
13.2.2 Resolving ambiguities 47
13.2.3 Problems with end-of-file newlines 48
13.3 Tagged file layout 49
13.4 Nonregular file layout 51
13.4.1 Example of database in nonregular file layout 52
13.5 What happens when a database is read in from disk 54

14 How information is displayed 57

14.1 Display specifications 57
14.2 Predefined displaytypes 59
14.2.1 Date displaytype 60
14.2.2 Time displaytype 60
14.3 Enumeration types 60
14.3.1 One-character enumeration displaytypes 61
14.3.2 Multi-character enumeration displaytypes 61
14.4 Defining new displaytypes 62
14.5 Display specification optional parameters 63
14.6 Display specification abbreviations 66

15 Customization 67

15.1 Auxiliary and format files 67
15.2 Hooks and customization functions 68
15.2.1 Load and read hooks 69
15.2.2 Database minor mode hooks 70
15.2.3 Record display hooks 70
15.2.4 Edit mode hooks 71
15.2.5 Display format change hooks 72
15.2.6 Recordfieldspec change hooks 75
15.3 Local variables 75
15.3.1 Per-data-display-buffer variables 75
15.3.2 Per-database variables 75
15.4 Global variables 76

16 Database representation 78

16.1 The database structure 78
16.1.1 The link structure 83
16.2 Mapping over the database 83
16.3 Manipulating records 85
16.3.1 Creating and copying records 86
16.3.2 Accessing record fields 86
16.3.3 Mapping over record fields 88

17 Naming conventions 89

17.1 Function and variable naming conventions 89
17.2 File naming conventions 89

18 EDB support 91

18.1 Updates to EDB 91
18.2 In case of trouble 92
18.2.1 Variables 92
18.2.2 Exiting Emacs or saving files 92
18.2.3 Compiling EDB 93
18.2.3.1 Load EDB before compiling it 93
18.2.4 Debugging EDB 94
18.2.4.1 Enabling debugging messages 94
18.2.4.2 Printing circular structures 95
18.2.5 Reporting bugs 96

Function Index 98

Variable Index 101

Concept Index 103