

This program is distributed under the terms of the GNU General Public License.

Copyright 2005, 2006, 2007 David Joseph Stith.

We shall present a simple CGI application that offers transparent view, update, and insert access in a truly relational way to any SQLite database that adheres to a few simple conventions. These conventions are:

- i. The first field on any table should have the same name as the table and should be created as INTEGER PRIMARY KEY.
- ii. Any other field having the same name as a foreign table should be a foreign key for that table. That's it!

1. Here is The Big Picture.

```

<Included files 2>
<Global variables 3>
<Fundamental procedures and structures 16>
<Callback procedures 22>
<Procedures 20>
<Main CGI method 5>

```

2. We will, of course, need the SQLite runtime library, and we will use CGIC to simplify the CGI interface. To list saved queries (stored as files) we need *dirent.h*.

```

<Included files 2> ≡
#include <stdio.h>
#include <stdlib.h>
#include <sqlite.h>
#include <cgic.h>
#include <dirent.h>

```

This code is used in section 1.

3. We shall only need one database, so the pointer to it may as well be global. Similarly, we need only handle one error at a time. Consequently, we can simplify our code with a simple SQLite_EXEC macro.

```

#define SQLite_EXEC(SQL, CALLBACK, ARG)
    error_code = sqlite_exec(database, SQL, CALLBACK, ARG, &error_text)
<Global variables 3> ≡
struct sqlite *database = Λ;
int error_code = 0; /* The integer error code returned by sqlite_open or sqlite_exec */
char *error_text = Λ; /* The human readable error message */

```

See also sections 12, 19, 27, 34, 39, 43, 58, 62, and 68.

This code is used in section 1.

4. Giving up when a serious SQL error occurs will have to suffice so long as we tell the world about it first. Most of the time, we will be giving up when *error_code* ≠ SQLite_OK. Hence,

```

<Give up if there was any SQL error 4> ≡
if (error_code ≠ SQLite_OK) {
    fprintf(cgiOut, "SQLite error: %s\n", error_text);
    <End HTML document 128>;
    <Close database 15>;
    exit(0);
}

```

This code is used in sections 14, 21, 35, 51, 65, 66, 85, 94, and 95.

5. We will protect our databases with a password, stored as a cookie. Only when the correct password is given may we proceed. We must be careful to set cookies before any other content. Here goes:

```

⟨Main CGI method 5⟩ ≡
int cgiMain(void)
{
    int password_is_correct = 0;
    ⟨Check password 6⟩;
    if (password_is_correct) {
        ⟨Set password cookie 7⟩;    /* By setting the password cookie every time, we insure that the
            expiration time of the cookie gets updated. */
        ⟨Set content type 115⟩;
        ⟨Begin HTML document 116⟩;
        ⟨Respond to valid user 9⟩;
    }
    else {
        ⟨Set content type 115⟩;
        ⟨Begin HTML document 116⟩;
        ⟨Ask for password 8⟩;
    }
    ⟨End HTML document 128⟩;
    return 0;
}

```

This code is used in section 1.

6. If the password cookie already exists and is correct then all is well and good. Otherwise, perhaps the user has just now submitted the password. Change this password in the change file ‘cgisqlite.ch’ before you compile this program!

```

#define PASSWORD "password1"
#define PASSWORD_LEN 10    /* Including the NULL char at the end. */
⟨Check password 6⟩ ≡
char password_given[PASSWORD_LEN];
int cgi_result = cgiCookieString("password", password_given, PASSWORD_LEN);
if (cgi_result ≡ cgiFormSuccess ∧ ¬strcmp(password_given, PASSWORD)) {
    password_is_correct = 1;
}
else {
    cgi_result = cgiFormString("password", password_given, PASSWORD_LEN);
    if (cgi_result ≡ cgiFormSuccess ∧ ¬strcmp(password_given, PASSWORD)) {
        password_is_correct = 1;
    }
}
}

```

This code is used in section 5.

7. This password cookie should be valid for one day (43200 seconds).

```

⟨Set password cookie 7⟩ ≡
    cgiHeaderCookieSetString("password", PASSWORD, 43200, cgiScriptName, cgiServerName);

```

This code is used in section 5.

8. Of course we must give them the opportunity to enter the password in the first place.

⟨Ask for password 8⟩ ≡

```
fprintf (cgiOut, "<h3>Welcome!</h3>\n"
        "You_must_enter_the_password_correctly_to_proceed.<br>"
        "Please_enter_the_password:\n"
        "<form_method=\"POST\"_action=\"%s%s\">\n"
        "<input_name=\"password\"_type=\"password\">\n"
        "<input_type=\"submit\"_value=\"enter\">\n"
        "</form>\n", cgiScriptName, cgiPathInfo);
```

This code is used in section 5.

9. We shall find it convenient to use CGI PATH_INFO and the CGI GET and POST variables in the following way: The PATH_INFO must provide the file path to the SQLITE database. A ‘s.query’ GET variable indicates that direct SQL access should be provided. The value will indicate the name of a SQL query to load. Otherwise we provide abstracted SQL access using the following variables: An optional ‘s.table’ GET variable may provide the name of the table to browse or update. Any GET variable ‘key’ (with no ‘.’) shall specify an integer value for the primary key named *key* to which the results should be limited. Any FORM variable ‘u.field’ shall specify a new value for the field *field* of the row keyed by the value of ‘u.table’.

```
#define TABLE_VAR "s.table"
#define TABLE_LEN 50 /* Maximum length for the value of the 's.table' variable */
#define QUERY_LEN 50
```

⟨Respond to valid user 9⟩ ≡

```
char *filepath;
char table[TABLE_LEN];
char query[QUERY_LEN];
⟨Get required filepath 13⟩;
⟨Open or create database 14⟩;
⟨Display 'home' link 25⟩;
if (parseQueryString("s.query", query, QUERY_LEN) ≡ cgiFormSuccess) {
    ⟨Provide direct SQL access 88⟩;
}
else {
    ⟨Display 'query' link 24⟩;
    ⟨Provide abstracted SQL access 10⟩;
}
⟨Close database 15⟩;
```

This code is used in section 5.

10. For our abstracted SQL access, we need only to provide three types of responses:

- i. If the 's.table' is absent, then we provide a link for every table in the database to browse (*a la* 'iii' below).
- ii. If the current 's.table' is restricted to a given primary key, then we provide the means to update the row or to browse other tables that contain the key as a foreign key. We shall let an empty (but present) value for the primary key indicate that we should provide the means to insert a new row. Submitting the update returns to 'iii' below.
- iii. Otherwise, we list the data in the table, limiting the rows according to the restrictions given, providing a submit button for each primary key that restricts results to that key (*a la* 'ii' above). The updates submitted by 'ii' above are handled here. The display is improved if we detail the query context imposed by the query string restrictions separately rather than repeating them redundantly within a single table of results.

```

⟨Provide abstracted SQL access 10⟩ ≡
  ⟨Copy cgiQueryString 69⟩;
  ⟨Populate tables list 21⟩;
  if (parseQueryString(TABLE_VAR, table, TABLE_LEN) ≠ cgiFormSuccess) {
    ⟨Display HTML horizontal rule 117⟩;
    ⟨Display tables list 23⟩;
  }
  else {
    ⟨Display 'tables' link 11⟩;
    ⟨Display HTML horizontal rule 117⟩;
    ⟨If necessary, perform updates 64⟩;
    if (*primary_key ≠ '\0' ∨ parseQueryString(table, primary_key, KEY_LEN) ≡ cgiFormSuccess) {
      ⟨Unless form is for INSERT, display drilldown links 84⟩;
      ⟨Display HTML horizontal rule 117⟩;
      ⟨Display update form 51⟩;
    }
    else {
      ⟨Display query context 50⟩;
      ⟨Display HTML horizontal rule 117⟩;
      ⟨Browse table 26⟩;
    }
  }
}

```

This code is used in section 9.

11. If there is anything more than the `TABLE_VAR` on the query string, then a ‘tables’ link is worthwhile to carry query limits into whatever table the user may choose. In this case an ampersand will definitely exist on the query string and the last query variable will be the `TABLE_VAR` which should be dropped from the link.

```

<Display ‘tables’ link 11> ≡
{
  char *new_query_string;
  char *last_ampersand;
  DUPLICATE_STRING(new_query_string, effective_query);
  last_ampersand = strchr(new_query_string, '&');
  if (last_ampersand) {
    *last_ampersand = '\0';
    fprintf(cgiOut, "<a href=\"%s%s?%s\">tables</a>\n", cgiScriptName, cgiPathInfo,
            new_query_string);
  }
}

```

This code is used in section 10.

12. We make the primary key a global variable so that callback procedures that create the HTML form for updating or inserting data can tell which kind to make. Recall that when a table name is given and the `primary_key` is empty, an `INSERT` is called for.

```

#define KEY_LEN 20
<Global variables 3> +≡
  char primary_key[KEY_LEN]; /* If present for the same table as the ‘s.table’ variable. */

```

13. If the filepath was not supplied by the user then we can do nothing.

```

<Get required filepath 13> ≡
  if (cgiPathInfo ≡ Λ ∨ cgiPathInfo[0] ≡ '\0') {
    fprintf(cgiOut, "You must specify the path to a SQLite database.\n</html>");
    exit(0);
  }
  else {
    filepath = &cgiPathInfo[1];
  }

```

This code is used in section 9.

14. We shall assume that file and directory security is sufficient that we can simply request to open the given filename as a SQLITE database. We will be needing to obtain column names from empty resultsets, so we set the PRAGMA accordingly. Finally, we allow a one second timeout before allowing SQLITE to declare a SQLITE_BUSY error.

```

⟨Open or create database 14⟩ ≡
    database = sqlite_open(filepath, 0, &error_text);
    if (!database) /* Something bad happened... */
    {
        fprintf(cgiOut, "%s\n", error_text);
        ⟨End HTML document 128⟩;
        exit(0);
    }
    SQLITE_EXEC("PRAGMA_empty_result_callbacks=1", Λ, Λ);
    ⟨Give up if there was any SQL error 4⟩;
    sqlite_busy_timeout(database, 1000);

```

This code is used in section 9.

15. Closing the database will be a simple matter.

```

⟨Close database 15⟩ ≡
    if (database) sqlite_close(database);

```

This code is used in sections 4 and 9.

16. Given the field naming conventions we have assumed, creating a SQL statement will be considerably simplified. Nevertheless, we shall still need to identify which tables to join. Let's use a list of strings to keep track of table names.

```

⟨Fundamental procedures and structures 16⟩ ≡
    struct list {
        char *string;
        struct list *next;
    };

```

See also sections 17, 18, 33, 36, 44, 82, and 105.

This code is used in section 1.

17. We will agree to always use freshly allocated strings in our list so that we can free them all like this:

```

⟨Fundamental procedures and structures 16⟩ +≡
    void free_list(list)
        struct list *list;
    {
        struct list *next;
        while (list ≠ Λ) {
            next = list->next;
            free(list->string);
            free(list);
            list = next;
        }
    }

```

18. Here is a convenient procedure to copy a string onto the head of the list.

```
#define NEW_INSTANCE(TYPE) (TYPE *) malloc(sizeof (TYPE))
#define DUPLICATE_STRING(DESTINATION, SOURCE)
    {
        DESTINATION = (char *) malloc(strlen(SOURCE) + 1);
        strcpy(DESTINATION, SOURCE);
    }
```

⟨Fundamental procedures and structures 16⟩ +≡

```
void copy_string_to_list(string, list_pointer)
    char *string;
    struct list **list_pointer;
{
    struct list *head = NEW_INSTANCE(struct list);
    head->next = *list_pointer;
    DUPLICATE_STRING(head->string, string);
    *list_pointer = head;
}
```

19. We will often need to check whether a table with a given name exists. Let's keep a list for ourselves rather than calling SQLITE every time we need to do this.

⟨Global variables 3⟩ +≡

```
struct list *tables = Λ; /* The list of all tables */
```

20. Here, then, is the procedure to look through a list.

⟨Procedures 20⟩ ≡

```
int string_exists(string, lst)
    char *string;
    struct list *lst;
{
    while (lst ≠ Λ) {
        if (¬strcmp(string, lst->string)) return 1; /* The string does exist */
        lst = lst->next;
    }
    return 0; /* The string does not exist */
}
```

See also section 35.

This code is used in section 1.

21. And here is how we populate the tables list:

```
#define SHOW_TABLES
    "SELECT name FROM sqlite_master WHERE type='table' ORDER BY name DESC"
```

⟨Populate tables list 21⟩ ≡

```
SQLITE_EXEC(SHOW_TABLES, compile_tables_list, Λ);
⟨Give up if there was any SQL error 4⟩;
```

This code is used in section 10.

22. The callback simply copies the table names into the *tables* list.

```

⟨ Callback procedures 22 ⟩ ≡
  int compile_tables_list(pArg, argc, argv, columnNames)
    void *pArg;
    int argc;
    char **argv;
    char **columnNames;
  {
    if (argv) {
      copy_string_to_list(argv[0], &tables);
    }
    return 0; /* This tells SQLITE to keep going. */
  }

```

See also sections 28, 37, 52, 86, 96, and 99.

This code is used in section 1.

23. Now let's implement the display of the list of tables. We must simply display a link for each table name in the *tables* list.

```

⟨ Display tables list 23 ⟩ ≡
  struct list *iterator = tables;
  ⟨ Begin HTML table 118 ⟩;
  ⟨ Begin table row 119 ⟩;
  ⟨ Begin table header 121 ⟩;
  fprintf(cgiOut, "tables");
  ⟨ End table header 122 ⟩;
  ⟨ End table row 126 ⟩;
  while (iterator) {
    ⟨ Begin table row 119 ⟩;
    ⟨ Begin table cell 123 ⟩;
    fprintf(cgiOut, "<a_href=\"%s%s?%s%s\"TABLE_VAR\"=%s\">%s</a>", cgiScriptName, cgiPathInfo,
      cgiQueryString, cgiQueryString[0] ≡ '\0' ? "" : "&", iterator->string, iterator->string);
    ⟨ End table cell 125 ⟩;
    ⟨ End table row 126 ⟩;
    iterator = iterator->next;
  }
  ⟨ End HTML table 127 ⟩;

```

This code is used in section 10.

24. Here is the link to obtain direct SQL access.

```

⟨ Display 'query' link 24 ⟩ ≡
  fprintf(cgiOut, "<a_href=\"%s%s?s.query=\">query</a>\n", cgiScriptName, cgiPathInfo);

```

This code is used in section 9.

25. Lastly, and probably least importantly, here is our courteous 'home' link to get back to the list of tables with all limits cleared (a blank query string).

```

⟨ Display 'home' link 25 ⟩ ≡
  fprintf(cgiOut, "<a_href=\"%s%s\">home</a>\n", cgiScriptName, cgiPathInfo);

```

This code is used in section 9.

26. Table Browsing. Alright, now we tackle the table browser.

```
#define SQL_BUFFER_LENGTH 3000
<Browse table 26> ≡
char sql_buffer[SQL_BUFFER_LENGTH];
create_sql_for_table(table, sql_buffer, SQL_BUFFER_LENGTH);
<Begin HTML table 118>;
display_column_names = 1;
/* Indicate to the callback that the column names need to be displayed */
interactive = 1; /* Indicate to the callback that we want hyperlinks */
SQLITE_EXEC(sql_buffer, table_browser_callback, table);
/* Note that the table name is passed to the callback */
<Display link for new row 60>;
<End HTML table 127>;
```

This code is used in sections 10 and 63.

27. So we need these global variables:

```
<Global variables 3> +≡
int display_column_names;
int interactive = 0;
```

28. Our callback displays each row as an HTML table row.

```
<Callback procedures 22> +≡
int table_browser_callback(pArg, argc, argv, columnNames)
void *pArg;
int argc;
char **argv;
char **columnNames;
{
    <Display column names if we have not yet 29>;
    if (argv) {{Display argv array 31}}
    return 0; /* This tells SQLITE to keep going */
}
```

29. Displaying the column names is easy. We also make sure to clear the *display_column_names* so that we won't display the column names more than once.

```
<Display column names if we have not yet 29> ≡
if (display_column_names) {
    int i;
    <Begin table row 119>;
    for (i = 0; i < argc; ++i) {
        <Begin table header 121>;
        <Display columnNames[i] 30>;
        <End table header 122>;
    }
    display_column_names = 0;
    <End table row 126>;
}
```

This code is used in section 28.

30. We can improve the display slightly if we omit the table prefix from the field names of the current table. Recall that *pArg* is the table name.

```

<Display columnNames[i] 30> ≡
  int table_name_len = strlen(pArg);
  if (¬strncmp(columnNames[i] + 1, pArg, table_name_len) ∧ columnNames[i][table_name_len + 1] ≡ '\')
  {
    cgiHtmlEscape(columnNames[i] + table_name_len + 3);
  }
  else {
    cgiHtmlEscape(columnNames[i]);
  }

```

This code is used in section 29.

31. When displaying the *argv* array, however, we must treat the primary key differently than the remaining fields if *interactive* ≡ 1.

```

<Display argv array 31> ≡
  int i;
  if (interactive) {
    <Begin row with primary key 59>;
  }
  else {
    <Begin table row 119>;
  }
  for (i = interactive; i < argc; ++i) {
    <Begin table cell 123>;
    <Display argv[i] 32>;
    <End table cell 125>;
  }
  <End table row 126>;

```

This code is used in section 28.

32. The only trick to displaying each entry in the *argv* array is that the value may be null. But let's add a hack to allow display of html directly from the value when the column name starts with 'html'.

```

<Display argv[i] 32> ≡
  if (argv[i]) {
    register char *name = columnNames[i];
    if (name[0] ≡ 'h' ∧ name[1] ≡ 't' ∧ name[2] ≡ 'm' ∧ name[3] ≡ 'l') {
      fprintf(cgiOut, argv[i]);
    }
    else {
      cgiHtmlEscape(argv[i]);
    }
  }
  else {
    <Display HTML NULL 124>;
  }

```

This code is used in section 31.

33. Now, to prepare ourselves for constructing the SQL statements we need, let's arm ourselves with a procedure for appending strings to a fixed-length buffer. This procedure should advance a pointer to the end of the buffer and decrease the length so that things stand ready for the next call to use the remainder safely. Therefore, the pointer to the end of the buffer and the length remaining must both be passed by reference. We will use two macros to make this look simpler.

```
#define BUFFER_END (*buffer_end_ref)
#define LENGTH_REMAINING (*length_remaining_ref)
<Fundamental procedures and structures 16> +≡
void buffer_append(buffer_end_ref, length_remaining_ref, string)
    char *BUFFER_END;
    int LENGTH_REMAINING;
    char *string;
{
    char *string_remaining = string;
    while (string_remaining[0] ≠ '\0') {
        if (LENGTH_REMAINING ≤ 1) {
            fprintf(cgiOut, "Buffer_Overflow!");
            exit(0);
        }
        *(BUFFER_END++) = *(string_remaining++);
        --LENGTH_REMAINING;
    }
    *BUFFER_END = '\0';
}
```

34. Since we will be dealing with SQLITE callbacks, it is much simpler to use global variables for our buffer paraphernalia.

```
#define WHERE_BUFFER_LEN 2000
<Global variables 3> +≡
struct list *visited = Λ; /* The tables we have already joined */
char *select_buffer_end;
int select_buffer_length;
char *from_buffer_end;
int from_buffer_length;
char *where_buffer_end;
int where_buffer_length;
```

35. Okay, here is the idea: Any field names in a query that exist in the database also as table names should be joined exactly once. We can get the field list for any table by examining the column names returned by an empty SQL query.

```
#define SHOW_FIELDS_LEN 1000
<Procedures 20> +≡
void join_tables(table)
    char *table;
{
    char show_fields[SHOW_FIELDS_LEN];
    snprintf(show_fields, SHOW_FIELDS_LEN, "SELECT_*_FROM_%s'_WHERE_1=0", table);
    SQLITE_EXEC(show_fields, join_tables_callback, table);
    <Give up if there was any SQL error 4>;
}
```

36. Since the *join_tables* procedure is itself called by its callback procedure, we must declare it prior to both.

```
<Fundamental procedures and structures 16> +≡
void join_tables(char *table);
```

37. Here is the callback, which iterates through the *columnNames*, dispatching each to the proper place to create the pieces of our SQL SELECT statement.

```
<Callback procedures 22> +≡
int join_tables_callback(pArg, argc, argv, columnNames)
    void *pArg;
    int argc;
    char **argv;
    char **columnNames;
{
    int i = -1;
    while (++i < argc) {
        char *field = columnNames[i];
        <Dispatch field for SQL creation 38>;
    }
    return 0; /* This tells SQLITE to keep going */
}
```

38. For the table's primary key, which must by convention be the first field, we apply any restriction that exists to the WHERE clause. But for each field except the first we check to see if the field name also exists as a table name. If it does and it has not already been joined, then we append the table join to the FROM clause then repeat the process for the foreign table. If it does not then we simply append the field to the SELECT clause.

```
<Dispatch field for SQL creation 38> ≡
if (i ≡ 0) /* Such that field is the primary key for the table */
{
    <Append restriction to where_buffer_end 42>;
}
else if (string_exists(field, tables)) {
    if (!string_exists(field, visited)) {
        copy_string_to_list(field, &visited);
        <Append join to from_buffer_end 41>;
        join_tables(visited→string);
    }
}
else {
    <Append field to SELECT clause 40>;
}
```

This code is used in section 37.

39. By counting how many fields are added to the SELECT clause, we can construct an ORDER BY clause using field numbers rather than names.

```
<Global variables 3> +≡
int select_field_count;
```

40. We will always have at least one field already in our SELECT clause, so that we can append the ‘,’ first.

```
#define APPEND_TO_SELECT(String) buffer_append(&select_buffer_end, &select_buffer_length, String)
< Append field to SELECT clause 40 > ≡
APPEND_TO_SELECT(", ");
APPEND_TO_SELECT(pArg); /* pArg is the table name */
APPEND_TO_SELECT("'. ');
APPEND_TO_SELECT(field);
APPEND_TO_SELECT("");
++select_field_count;
```

This code is used in section 38.

41. To join a table, we append the join clause to the *from_buffer* for the table of the same name as this field.

```
#define APPEND_TO_FROM(String) buffer_append(&from_buffer_end, &from_buffer_length, String)
< Append join to from_buffer_end 41 > ≡
APPEND_TO_FROM("␣LEFT␣JOIN␣");
APPEND_TO_FROM(field);
APPEND_TO_FROM("␣ON␣");
APPEND_TO_FROM(pArg); /* pArg is the table name */
APPEND_TO_FROM("'. ');
APPEND_TO_FROM(field);
APPEND_TO_FROM("'= ');
APPEND_TO_FROM(field);
APPEND_TO_FROM("'. ');
APPEND_TO_FROM(field);
APPEND_TO_FROM("");
```

This code is used in section 38.

42. We have a restriction to append if the field exists as a CGI variable. If we do have a restriction to apply, we also **break** the loop in order to avoid including any fields from the restricted table in the SELECT statement. This we do because these fields must always have the same value which would be redundantly repeated in the displayed table. However, we wouldn't want to skip them on the very first table, so we activate *skip_redundant_fields* only after the first pass. For an added twist, we allow the existence of a *selected_radio_key* to indicate that we ought to include all results for the first table even when a restriction for it exists. Otherwise the user would be unable to change the currently selected key.

```
#define VALUE_LEN 50
#define APPEND_TO_WHERE(String) buffer_append(&where_buffer_end, &where_buffer_length, String)
⟨Append restriction to where_buffer_end 42⟩ ≡
char value[VALUE_LEN];
if ((skip_redundant_fields ∨ ¬selected_radio_key ∨ *selected_radio_key ≡ '\0') ∧ parseQueryString(field,
    value, VALUE_LEN) ≡ cgiFormSuccess) {
    APPEND_TO_WHERE(" AND ");
    APPEND_TO_WHERE(field);
    APPEND_TO_WHERE(" ' ' ");
    APPEND_TO_WHERE(field);
    APPEND_TO_WHERE(" '=' ");
    APPEND_TO_WHERE(value);
    APPEND_TO_WHERE(" ");
    if (skip_redundant_fields) break; /* Process no more fields for this table */
}
skip_redundant_fields = 1;
```

This code is used in section 38.

43. So, this *skip_redundant_fields* will have to be global.

```
⟨Global variables 3⟩ +≡
int skip_redundant_fields;
```

44. Now we can finally get down to business. We can use the *sql_buffer* given for our SELECT buffer since it constitutes the beginning of our SQL statement.

```
#define FROM_BUFFER_LENGTH 2000
#define WHERE_BUFFER_LENGTH 2000
<Fundamental procedures and structures 16> +≡
void create_sql_for_table(table, sql_buffer, sql_buffer_length)
    char *table;
    char *sql_buffer;
    int sql_buffer_length;
{
    char from_buffer[FROM_BUFFER_LENGTH];
    char where_buffer[WHERE_BUFFER_LENGTH];
    <Initialize sql_buffer 45>;
    <Initialize from_buffer 46>;
    <Initialize where_buffer 47>;
    copy_string_to_list(table, &visited);
    skip_redundant_fields = 0; /* I'll explain later! */
    join_tables(visited_string); /* Using our fresh new copy */
    <Append from_buffer and where_buffer to sql_buffer 48>;
    <Append ORDER BY clause to sql_buffer 49>;
    free_list(visited);
    visited = Λ;
}
```

45. The *sql_buffer* must be primed with the primary key for the table since *join_tables* will forever refrain from doing so.

```
<Initialize sql_buffer 45> ≡
select_buffer_end = sql_buffer;
select_buffer_length = sql_buffer_length;
select_field_count = 1;
APPEND_TO_SELECT("SELECT_");
APPEND_TO_SELECT(table);
APPEND_TO_SELECT("'. '");
APPEND_TO_SELECT(table);
APPEND_TO_SELECT("');
```

This code is used in section 44.

46. The *from_buffer* starts simply.

```
<Initialize from_buffer 46> ≡
from_buffer_end = from_buffer;
from_buffer_length = FROM_BUFFER_LENGTH;
APPEND_TO_FROM("_FROM_");
APPEND_TO_FROM(table);
APPEND_TO_FROM("');
```

This code is used in section 44.

47. We might not need a where clause, so we will save the appending of the word “ WHERE ” for later.

```

< Initialize where_buffer 47 > ≡
  where_buffer_end = where_buffer;
  where_buffer_length = WHERE_BUFFER_LENGTH;
  where_buffer[0] = '\0';

```

This code is used in section 44.

48. If there is any where clause to append, append it, but skip the first “ AND ”. We also insure that any *selected_radio_key* appears in the resultset so that it may, in fact, be selected.

```

< Append from_buffer and where_buffer to sql_buffer 48 > ≡
  APPEND_TO_SELECT(from_buffer);
  if (where_buffer[0] ≠ '\0') {
    APPEND_TO_SELECT(" WHERE ");
    APPEND_TO_SELECT(where_buffer + 5);
    APPEND_TO_SELECT(" ");
    if (selected_radio_key) {
      APPEND_TO_SELECT(" OR ");
      APPEND_TO_SELECT(table);
      APPEND_TO_SELECT(" . ");
      APPEND_TO_SELECT(table);
      APPEND_TO_SELECT(" = ");
      APPEND_TO_SELECT(selected_radio_key);
      APPEND_TO_SELECT(" ");
    }
  }

```

This code is used in section 44.

49. Our ORDER BY clause will simply be for fields 2 through 9.

```

< Append ORDER BY clause to sql_buffer 49 > ≡
  if (select_field_count > 1) {
    char field_number[3];
    field_number[0] = ' ';
    field_number[1] = '2';
    field_number[2] = '\0';
    APPEND_TO_SELECT(" ORDER BY ");
    if (--select_field_count > 8) select_field_count = 8;
    while (--select_field_count > 0) {
      ++field_number[1];
      APPEND_TO_SELECT(field_number);
    }
  }

```

This code is used in section 44.

50. Displaying the Query Context. We want to provide information about the chosen row of each table restricted by the query string variables.

```

⟨Display query context 50⟩ ≡
char sql_context_buffer[SQL_BUFFER_LENGTH];
char *query_pointer;
char *equal_pointer;
char *query_string_copy;
DUPLICATE_STRING(query_string_copy, cgiQueryString);
query_pointer = query_string_copy - 1;
while (query_pointer) {
    ++query_pointer;
    equal_pointer = strchr(query_pointer, '=');
    if (equal_pointer) {
        *equal_pointer = '\\0';
        if (string_exists(query_pointer, tables)) {
            create_sql_for_table(query_pointer, sql_context_buffer, SQL_BUFFER_LENGTH);
            ⟨Begin HTML table 118⟩;
            display_column_names = 1;
            /* Indicate to the callback that the column names need to be displayed */
            interactive = 0; /* Indicate to the callback that we do not want hyperlinks */
            SQLITE_EXEC(sql_context_buffer, table_browser_callback, query_pointer);
            /* Note that the table name is passed to the callback */
            ⟨End HTML table 127⟩;
        }
        query_pointer = equal_pointer + 1;
    }
    query_pointer = strchr(query_pointer, '&');
}
free(query_string_copy);

```

This code is used in section 10.

51. Displaying an Update Form. To update the row of our *table* whose key is *key*, we must provide HTML form input for each field, pre-selecting the current values.

```
#define SELECT_ROW_LEN 1000
< Display update form 51 > ≡
char select_row[SELECT_ROW_LEN];
snprintf(select_row, SELECT_ROW_LEN, "SELECT_*_FROM_'%s' WHERE_'%s' . '%s'='%s'", table, table,
table, primary_key);
SQLITE_EXEC(select_row, update_row_callback, table);
< Give up if there was any SQL error 4 >;
```

This code is used in section 10.

52. Of course the callback is where most everything gets done. We should ignore the case where *argv* is NULL unless the *primary_key* is empty (thus indicating that we are making a form for an INSERT request) since there will be no callback with *argv* data in that case.

```
#define INSERT_REQUESTED (*primary_key ≡ '\0')
< Callback procedures 22 > +≡
int update_row_callback(pArg, argc, argv, columnNames)
void *pArg;
int argc;
char **argv;
char **columnNames;
{
if (argv ∨ INSERT_REQUESTED) {
< Create update form 53 >;
}
else {
fprintf(cgiOut, "This_row_appears_no_longer_to_exist!");
}
return 0;
}
```

53. Let's frame our update form inside a table. If the form is cancelled, the results should no longer be limited to the primary key for this table. We know that the restriction for this table is the first one on the query string, so we can just lop off the beginning. We will need to indicate which row is to be updated by using the 'u.table' form variable.

```
< Create update form 53 > ≡
char *new_query_string;
< Locate new_query_string 54 >;
fprintf(cgiOut, "<form_method=\\\"POST\\\"_action=\\\"%s%s?%s\\\">\n", cgiScriptName, cgiPathInfo,
cgiQueryString);
fprintf(cgiOut, "<a_href=\\\"%s%s?%s\\\">close</a>\n", cgiScriptName, cgiPathInfo, new_query_string);
< Begin HTML table 118 >;
< Make form elements for fields 55 >;
< End HTML table 127 >;
fprintf(cgiOut, "<input_type=\\\"submit\\\"_value=\\\"apply\\\">\n"
"</form>\n");
```

This code is used in section 52.

54. Finding the first ampersand and adding one should do the trick. We must have an ampersand since we must have had the table specified on the query string as well.

```
<Locate new_query_string 54> ≡
    new_query_string = strchr(cgiQueryString, '&') + 1;
```

This code is used in section 53.

55. Now, for each field, except the primary key, we provide HTML form input as follows: If the field is a foreign key, we list the rows of the foreign table such that each row has a HTML radio button and the row for the current foreign key is preselected. Otherwise, we simply use a HTML text box.

```
<Make form elements for fields 55> ≡
    int i;
    for (i = 0; i < argc; ++i) {
        <Begin table row 119>;
        <Begin table header 121>;
        cgiHtmlEscape(columnNames[i]);
        <End table header 122>;
        <Begin table cell 123>;
        if (i == 0) {
            <Display the primary key for the update form 56>;
        }
        else if (string_exists(columnNames[i], tables)) {
            <Create radio buttons 63>;
        }
        else {
            <Create text entry 57>;
        }
        <End table cell 125>;
        <End table row 126>;
    }
}
```

This code is used in section 53.

56. The primary key should be displayed but not updated.

```
<Display the primary key for the update form 56> ≡
    fprintf(cgiOut, "<input type=\"hidden\" name=\"u.%s\" value=\"%s\">%s\n", pArg, primary_key,
            primary_key);
```

This code is used in section 55.

57. The text box entry is pretty simple. We have the name of the field in *columnNames[i]* and the current value in *argv[i]*. Let's give a generous 60 character size for the text box.

```
<Create text entry 57> ≡
    fprintf(cgiOut, "<input type=\"text\" size=\"60\" name=\"u.\"");
    cgiHtmlEscape(columnNames[i]);
    fprintf(cgiOut, "\" value=\"");
    if (argv & argv[i]) /* Either one may be NULL! */
        cgiHtmlEscape(argv[i]);
    fprintf(cgiOut, "\">\n");
```

This code is used in section 55.

58. We can reuse the table browser to create radio buttons by indicating the currently selected key in a global variable that otherwise will be NULL.

```
<Global variables 3> +≡
char *selected_radio_key = Λ;
```

59. Only the display of the primary key need differ. The table browser will set no *selected_radio_key* and will display the primary keys as links that restrict the results to that primary key's value.

```
<Begin row with primary key 59> ≡
if (selected_radio_key) {
    <Begin row with radio button 61>;
}
else {
    <Begin table row 119>;
    <Begin table cell 123>;
    fprintf (cgiOut, "<a href=\"%s%s?%s=%s&%s\">%s</a>", cgiScriptName, cgiPathInfo, pArg, argv[0],
             cgiQueryString, argv[0]); /* Remember that pArg is the table name passed to the callback */
}
<End table cell 125>;
```

This code is used in section 31.

60. Likewise, the link for insertion of a new row into the table is only applicable to the table browser. We will use a blank key value to indicate the INSERT request.

```
<Display link for new row 60> ≡
if (¬selected_radio_key) {
    <Begin table row 119>;
    <Begin table cell 123>;
    fprintf (cgiOut, "<a href=\"%s%s?%s=&%s\">New</a>", cgiScriptName, cgiPathInfo, table,
             cgiQueryString);
    <End table cell 125>;
    <End table row 126>;
}
```

This code is used in section 26.

61. The update form requires the proper radio button to be pre-selected and highlit. For an insert, *select_first_radio* will be initially set so that we can insure that the first, and only the first, radio button is pre-selected but not highlit.

```

⟨Begin row with radio button 61⟩ ≡
  if (select_first_radio) {
    select_first_radio = 0;
    ⟨Begin table row 119⟩;
    ⟨Begin table cell 123⟩;
    fprintf(cgiOut, "<input type=\"radio\" checked");
  }
  else if (argv & !strcmp(selected_radio_key, argv[0])) {
    ⟨Begin highlit table row 120⟩;
    ⟨Begin table cell 123⟩;
    fprintf(cgiOut, "<input type=\"radio\" checked");
  }
  else {
    ⟨Begin table row 119⟩;
    ⟨Begin table cell 123⟩;
    fprintf(cgiOut, "<input type=\"radio\"");
  }
  fprintf(cgiOut, " name=\"%u.%s\" value=\"%s\">%s", pArg, argv[0], argv[0]);

```

This code is used in section 59.

62. This global variable allows us to insure that the first and only the first radio button is selected when displaying the update form for a new record.

```

⟨Global variables 3⟩ +≡
  int select_first_radio = 0;

```

63. Now we can use the same code as the table browser so long as we set the *selected_radio_key*. We make our own copies since the *argv[i]* and *columnNames[i]* we have may not survive additional calls to SQLITE. But if *argv* or *argv[i]* is NULL then an empty string as the *selected_radio_key* must suffice to prevent any radio buttons from being pre-selected. When inserting a new row, however, we should select the first radio button just to insure that the user cannot leave the field NULL.

```

⟨Create radio buttons 63⟩ ≡
  char *table;
  DUPLICATE_STRING(table, columnNames[i]);
  if (argv & argv[i]) {
    DUPLICATE_STRING(selected_radio_key, argv[i]);
  }
  else {
    selected_radio_key = "";
    if (!argv) /* Inserting new row */
      select_first_radio = 1;
  }
  ⟨Browse table 26⟩;
  if (*selected_radio_key != '\0') free(selected_radio_key);
  selected_radio_key = Λ;
  free(table);

```

This code is used in section 55.

64. When we receive the form submission, we will know by the presence of a form field 'u.table' for the current *table*. The value of this form field is the key we must update. If the value is empty, however, it is an INSERT we must perform.

```
#define UPDATE_KEY_VALUE_LEN 50
#define UPDATE_KEY_NAME_LEN 50
<If necessary, perform updates 64> ≡
char update_key_name[UPDATE_KEY_NAME_LEN];
char update_key_value[UPDATE_KEY_VALUE_LEN];
snprintf(update_key_name, UPDATE_KEY_NAME_LEN, "u.%s", table);
int cgi_result = cgiFormString(update_key_name, update_key_value, UPDATE_KEY_VALUE_LEN);
if (cgi_result ≡ cgiFormSuccess) {
    <Perform updates 65>;
}
else if (cgi_result ≡ cgiFormEmpty) {
    <Perform insert 66>;
}
```

This code is used in section 10.

65. This time we can use our own automatic variables to build our SQL UPDATE statement.

```
#define UPDATE_BUFFER_LEN 10000
#define APPEND_TO_UPDATE(String) buffer_append(&update_buffer_end, &update_buffer_length, String)
<Perform updates 65> ≡
char update_buffer[UPDATE_BUFFER_LEN];
char *update_buffer_end = update_buffer;
int update_buffer_length = UPDATE_BUFFER_LEN;
<Build SQL UPDATE statement 71>;
SQLITE_EXEC(update_buffer, Λ, Λ);
<Give up if there was any SQL error 4>;
<Report successful update 70>;
```

This code is used in section 64.

66. Likewise for the SQL INSERT, but it must be built in two parts.

```
#define INSERT_BUFFER_LEN 10000
#define INSERT_VALUES_LEN 8000
#define APPEND_TO_INSERT(String) buffer_append(&insert_buffer_end, &insert_buffer_length, String)
#define APPEND_TO_VALUES(String) buffer_append(&insert_values_end, &insert_values_length, String)
<Perform insert 66> ≡
char insert_buffer[INSERT_BUFFER_LEN];
char *insert_buffer_end = insert_buffer;
int insert_buffer_length = INSERT_BUFFER_LEN;
char insert_values[INSERT_VALUES_LEN];
char *insert_values_end = insert_values;
int insert_values_length = INSERT_VALUES_LEN;
<Build SQL INSERT statement 72>;
SQLITE_EXEC(insert_buffer, Λ, Λ);
<Give up if there was any SQL error 4>;
<Pretend that we are updating the new row 67>;
<Report successful update 70>;
```

This code is used in section 64.

67. After an insert we want to act exactly as if the user had chosen to update the row.

```

⟨Pretend that we are updating the new row 67⟩ ≡
    printf(primary_key, "%d", sqlite_last_insert_rowid(database));
    {
        char *start = strchr(effective_query, '&');
        char *end = effective_query + strlen(effective_query);
        int offset = strlen(primary_key);
        char *i;
        int j;
        for (i = end; i ≥ start; --i) {
            *(i + offset) = *i;
        }
        for (j = 0; j < offset; ++j) {
            *(start + j) = primary_key[j];
        }
    }
}

```

This code is used in section 66.

68. Hence we need another global variable.

```

⟨Global variables 3⟩ +=
    char *effective_query;

```

69. Initially this is simply the *cgiQueryString*

```

⟨Copy cgiQueryString 69⟩ ≡
    effective_query = (char *) malloc(strlen(cgiQueryString) + 20);
    strcpy(effective_query, cgiQueryString);

```

This code is used in section 10.

70. Users like to know that their updates were not ignored.

```

⟨Report successful update 70⟩ ≡
    fprintf(cgiOut, "<font_color=\"red\">Your_changes_were_saved.</font><br>");

```

This code is used in sections 65 and 66.

71. We will be appending a comma to the end of each field/value pair in the SET portion of the statement, therefore we simply remove the last one before appending the WHERE clause.

```

⟨Build SQL UPDATE statement 71⟩ ≡
    APPEND_TO_UPDATE("UPDATE");
    APPEND_TO_UPDATE(table);
    APPEND_TO_UPDATE("' SET");
    ⟨Append fields to update 73⟩;
    --update_buffer_end; /* Remove last comma */
    APPEND_TO_UPDATE(" WHERE");
    APPEND_TO_UPDATE(table);
    APPEND_TO_UPDATE("' .");
    APPEND_TO_UPDATE(table);
    APPEND_TO_UPDATE("' =");
    APPEND_TO_UPDATE(update_key_value);
    APPEND_TO_UPDATE("");

```

This code is used in section 65.

72. Likewise, the INSERT statement:

```

<Build SQL INSERT statement 72> ≡
APPEND_TO_INSERT("INSERT INTO ");
APPEND_TO_INSERT(table);
APPEND_TO_INSERT(" ");
APPEND_TO_VALUES("VALUES");
<Append fields to insert 74>;
--insert_buffer_end; /* Remove last comma */
APPEND_TO_INSERT(insert_values);
*(--insert_buffer_end) = ')'; /* Replace last comma with closing parenthesis */

```

This code is used in section 66.

73. To identify the fields to update, We can either iterate through the fields of the table checking for form inputs with names like *'u.field'* or we can iterate through the form variables and ignore all that do not begin with *'u.'* Iterating through the form variables is probably more efficient, if perhaps less elegant. Let's choose the more efficient method.

```

<Append fields to update 73> ≡
char **form_entries;
if (cgiFormEntries(&form_entries) ≡ cgiFormSuccess) {
    <Iterate through form variables for UPDATE 75>;
    cgiStringArrayFree(form_entries);
}
else {
    <Abandon attempt to process update request 77>;
}

```

This code is used in section 71.

74. There is no reason for the INSERT to proceed differently.

```

<Append fields to insert 74> ≡
char **form_entries;
if (cgiFormEntries(&form_entries) ≡ cgiFormSuccess) {
    <Iterate through form variables for INSERT 76>;
    cgiStringArrayFree(form_entries);
}
else {
    <Abandon attempt to process update request 77>;
}

```

This code is used in section 72.

75. The *form_entries* conclude with a Λ pointer, making iteration easy. We must remember to ignore the key when we come across it.

```

⟨ Iterate through form variables for UPDATE 75 ⟩ ≡
int i = 0;
char *form_entry;
while (form_entry = form_entries[i++]) {
  if (form_entry[0] ≡ 'u' ^ form_entry[1] ≡ '.') {
    char *field = &form_entry[2];
    if (strcmp(field, table)) {
      ⟨ Add field and value to UPDATE statement 78 ⟩;
    }
  }
}

```

This code is used in section 73.

76. *Dèjà vu!*

```

⟨ Iterate through form variables for INSERT 76 ⟩ ≡
int i = 0;
char *form_entry;
while (form_entry = form_entries[i++]) {
  if (form_entry[0] ≡ 'u' ^ form_entry[1] ≡ '.') {
    char *field = &form_entry[2];
    if (strcmp(field, table)) {
      ⟨ Add field and value to INSERT statement 79 ⟩;
    }
  }
}

```

This code is used in section 74.

77. For now, we won't say why we failed to process an update or insert request. We will just apologize for having failed.

```

⟨ Abandon attempt to process update request 77 ⟩ ≡
fprintf(cgiOut, "I'm very sorry, but I failed to process your update request.</html>");
exit(0);

```

This code is used in sections 73, 74, 80, and 81.

78. Now we have a true updateable field to process.

```

⟨ Add field and value to UPDATE statement 78 ⟩ ≡
APPEND_TO_UPDATE("");
APPEND_TO_UPDATE(field);
APPEND_TO_UPDATE("'=");
⟨ Append value to UPDATE statement 80 ⟩;
APPEND_TO_UPDATE(",");

```

This code is used in section 75.

79. For the INSERT, we must append to the field list and values list separately.

```

<Add field and value to INSERT statement 79> ≡
APPEND_TO_INSERT("");
APPEND_TO_INSERT(field);
APPEND_TO_INSERT(",");
<Append value to INSERT statement 81>;
APPEND_TO_VALUES(",");

```

This code is used in section 76.

80. First we must get the value from the form, then append it. Let's be generous with the allowable length of the value. We will interpret an empty string to mean a SQL null.

```

#define UPDATE_VALUE_LEN 6000
<Append value to UPDATE statement 80> ≡
char update_value[UPDATE_VALUE_LEN];
cgi_result = cgiFormString(form_entry, update_value, UPDATE_VALUE_LEN);
if (cgi_result ≡ cgiFormSuccess) {
    APPEND_TO_UPDATE("");
    buffer_append_with_apostrophies(&update_buffer_end, &update_buffer_length, update_value);
    APPEND_TO_UPDATE("");
}
else if (cgi_result ≡ cgiFormEmpty) {
    APPEND_TO_UPDATE("null");
}
else {
    <Abandon attempt to process update request 77>;
}

```

This code is used in section 78.

81. And once again, for the INSERT statement:

```

#define INSERT_VALUE_LEN 6000
<Append value to INSERT statement 81> ≡
char insert_value[INSERT_VALUE_LEN];
cgi_result = cgiFormString(form_entry, insert_value, INSERT_VALUE_LEN);
if (cgi_result ≡ cgiFormSuccess) {
    APPEND_TO_VALUES("");
    buffer_append_with_apostrophies(&insert_values_end, &insert_values_length, insert_value);
    APPEND_TO_VALUES("");
}
else if (cgi_result ≡ cgiFormEmpty) {
    APPEND_TO_VALUES("null");
}
else {
    <Abandon attempt to process update request 77>;
}

```

This code is used in section 79.

82. Appending the value is nearly identical with the code for the *buffer_append* procedure, but apostrophies must be doubled in order to safely enclose them within the apostrophies that delimit strings in SQLite statements.

(Fundamental procedures and structures 16) +≡

```

void buffer_append_with_apostrophies (buffer_end_ref, length_remaining_ref, string)
    char *BUFFER_END;
    int LENGTH_REMAINING;
    char *string;
{
    char *string_remaining = string;
    while (string_remaining[0] ≠ '\0') {
        if (LENGTH_REMAINING ≤ 2) {
            fprintf (cgiOut, "Buffer_Overflow!");
            exit (0);
        }
        if ((*BUFFER_END++) = *string_remaining) ≡ '\') {
            *BUFFER_END++ = *string_remaining;    /* Again */
            --LENGTH_REMAINING;
        }
        string_remaining++;
        --LENGTH_REMAINING;
    }
    *BUFFER_END = '\0';
}

```

83. Displaying the Drilldown Links. Within the context of a single row from a given table, it makes sense to offer links to all tables that contain our primary key as a foreign key and to limit the results to this key.

In order to accomplish this, we must look through the field names of every other table for a field with the same name as the current table.

```

<Display drilldown links 83> ≡
struct list *table_iterator = tables;
while (table_iterator) {
    if (strcmp(table_iterator->string, table)) /* Ignore the current table */
    {
        <Search fields for foreign key for our table 85>;
    }
    table_iterator = table_iterator->next;
}

```

This code is used in section 84.

84. But if the row does not yet exist, then the drilldown links would be rendered meaningless.

```

<Unless form is for INSERT, display drilldown links 84> ≡
if (!INSERT_REQUESTED) {
    <Display drilldown links 83>;
}

```

This code is used in section 10.

85. Since the callback will have to do the job, we pass it the table name to look for among the fields.

```

#define SEARCH_FIELDS_LEN 1000
<Search fields for foreign key for our table 85> ≡
char search_fields[SEARCH_FIELDS_LEN];
snprintf(search_fields, SEARCH_FIELDS_LEN, "SELECT_*_FROM_'%s' WHERE_1=0", table_iterator->string);
SQLITE_EXEC(search_fields, search_fields_callback, table);
<Give up if there was any SQL error 4>;

```

This code is used in section 83.

86. Hence the callback looks for *pArg* among the *columnNames* and displays a link if a match is found. The first field can be ignored since that one should be the primary key for the foreign table and therefore cannot be a foreign key for our table.

```

<Callback procedures 22> +≡
int search_fields_callback(pArg, argc, argv, columnNames)
    void *pArg;
    int argc;
    char **argv;
    char **columnNames;
{
    while (--argc > 0) {
        if (!strcmp(columnNames[argc], pArg)) /* pArg is the current table name */
        {
            <Display link to foreign table 87>;
        }
    }
}

```

87. Our database conventions guarantee that the name of the first field is also the name of the table. We may also assume that the *s.table* query string variable occurs and occurs last, since we must have had it in order to reach this point, and we have in all places only prepended to the query string. Using these facts, we create our *pièce de résistance*, the foreign table link.

```
#define FOREIGN_TABLE columnNames[0]
⟨ Display link to foreign table 87 ⟩ ≡
char *new_query_string;
char *last_equal_sign;
DUPLICATE_STRING(new_query_string, effective_query);
last_equal_sign = strrchr(new_query_string, '=');
*last_equal_sign = '\0';
fprintf(cgiOut, "<a href=\"%s%s?%s=%s\">%s</a>\n", cgiScriptName, cgiPathInfo, new_query_string,
        FOREIGN_TABLE, FOREIGN_TABLE);
free(new_query_string);
break; /* There is no need to look through the remaining fields of the foreign table */
```

This code is used in section 86.

88. Direct SQL access. Here we want to give the user a textarea in which to type straight SQL. Then, the two actions offered for our direct SQL access form will be:

- i. To execute the current SQL statement.
- ii. To save the current SQL statement to a file. In order to save queries, a directory with the same name as the database followed by ‘_cgisqlite’ should already exist. This is where we will store queries as simple ascii files.

```
#define RAW_SQL_LEN 8000
#define PARAMETER_SQL_LEN 8000
#define QUERY_NAME_LEN 100
#define PARAMETER_LEN 50
#define ACTION_LEN 30
<Provide direct SQL access 88> ≡
char query_path[QUERY_PATH_LEN];
char parameter_sql[PARAMETER_SQL_LEN];
<Display HTML horizontal rule 117>;
strncpy(query_path, filepath, QUERY_PATH_LEN);
strncat(query_path, "_cgisqlite/", QUERY_PATH_LEN);
<Read direct SQL form variables 89>;
if (!strcmp(action, "Save")) {
    <Save raw_sql 102>;
}
<Display saved query links 98>;
<Display HTML horizontal rule 117>;
<Display form for direct SQL access 90>;
<Display HTML horizontal rule 117>;
<Parameterize sql statement 93>;
if (!strcmp(action, "Execute_tabulated")) {
    <Execute parameter_sql and write HTML table 94>;
}
else if (!strcmp(action, "Execute_unformatted")) {
    <Execute parameter_sql and write unformatted 95>;
}
```

This code is used in section 9.

89. Here are all of the form variable we need.

```
<Read direct SQL form variables 89> ≡
char raw_sql[RAW_SQL_LEN];
char query_name[QUERY_NAME_LEN] = "";
char parameter[PARAMETER_LEN] = "";
char action[ACTION_LEN];
cgiFormString("sql", raw_sql, RAW_SQL_LEN);
cgiFormString("query_name", query_name, QUERY_NAME_LEN);
cgiFormString("parameter", parameter, PARAMETER_LEN);
cgiFormString("action", action, ACTION_LEN);
```

This code is used in section 88.

90. Our form should offer:

- i. Means to edit and to execute the current SQL query.
- ii. Means to enter a parameter value.
- iii. Means to enter a filename and to save the current SQL query there.

⟨Display form for direct SQL access 90⟩ ≡

```
fprintf (cgiOut, "<form_method=\"POST\"_action=\"%s%s?s.query=\">\n", cgiScriptName, cgiPathInfo);
⟨Display SQL form input 97⟩;
⟨Display parameter input 91⟩;
⟨Display saved query filename input 92⟩;
fprintf (cgiOut, "</form>\n");
```

This code is used in section 88.

91. The second of these is no more than a text input for the parameter value. We retain the *parameter* submitted to us.

⟨Display parameter input 91⟩ ≡

```
fprintf (cgiOut, "with_parameter:<input_type=\"text\"_name=\"parameter\"_value=\"");
cgiHtmlEscape (parameter);
fprintf (cgiOut, "\">\n");
```

This code is used in section 90.

92. The third of these is no more than a text input for the filename and a submit button. We retain the *query_name* submitted to us.

⟨Display saved query filename input 92⟩ ≡

```
fprintf (cgiOut, "or_save_as:<input_type=\"text\"_name=\"query_name\"_value=\"");
if (query[0] ≡ '\0') {
    cgiHtmlEscape (query_name);
}
else {
    cgiHtmlEscape (query);
}
fprintf (cgiOut, "\">\n<input_type=\"submit\"_name=\"action\"_value=\"Save\">\n");
```

This code is used in section 90.

93. Parameterizing the *raw_sql* statement is simplified by the use of '%s' as the replaced text. This will work so long as there are not more than thirteen occurrences.

⟨Parameterize sql statement 93⟩ ≡

```
snprintf (parameter_sql, PARAMETER_SQL_LEN, raw_sql, parameter, parameter, parameter, parameter,
parameter, parameter, parameter, parameter, parameter, parameter, parameter, parameter);
```

This code is used in section 88.

94. Our *table_browser_callback* will suffice to display the results of our *parameter_sql* statement as an HTML table.

⟨Execute *parameter_sql* and write HTML table 94⟩ ≡

```
⟨Begin HTML table 118⟩;
display_column_names = 1;
SQLITE_EXEC (parameter_sql, table_browser_callback, "");
⟨End HTML table 127⟩;
⟨Give up if there was any SQL error 4⟩;
```

This code is used in section 88.

95. For unformatted results, we will simply spit out the results, only escaping for HTML special characters and writing a terminating newline character.

```

<Execute parameter_sql and write unformatted 95> ≡
    SQLITE_EXEC(parameter_sql, unformatted_callback, Λ);
    fprintf(cgiOut, "\n");
    <Give up if there was any SQL error 4>;

```

This code is used in section 88.

96. Here is the callback for the unformatted results.

```

<Callback procedures 22> +≡
    int unformatted_callback(pArg, argc, argv, columnNames)
        void *pArg;
        int argc;
        char **argv;
        char **columnNames;
    {
        if (argv) {
            int i;
            for (i = 0; i < argc; ++i) {
                if (argv[i]) cgiHtmlEscape(argv[i]);
            }
        }
        return 0; /* This tells SQLITE to keep going. */
    }

```

97. Unless the user has requested to load a saved query, we will retain the SQL statement last submitted in the textarea both so that the user can modify it and so the user can see the SQL that was executed. Also, the parameter last used should be retained.

```

<Display SQL form input 97> ≡
    fprintf(cgiOut, "<textarea_name=\"sql\"_cols=\"80\"_rows=\"5\">\n");
    if (query[0] ≡ '\0') {
        cgiHtmlEscape(raw_sql);
    }
    else {
        <Load requested query 100>;
    }
    fprintf(cgiOut, "</textarea><br>\n"
        "<input_type=\"submit\"_name=\"action\"_value=\"Execute_tabulated\">\n"
        "<input_type=\"submit\"_name=\"action\"_value=\"Execute_unformatted\">\n");

```

This code is used in section 90.

98. Providing links for saved queries is little more than scanning the directory at *query_path*.

```
#define QUERY_PATH_LEN 100
<Display saved query links 98> ≡
    struct dirent **queries;
    int n;
    n = scandir(query_path, &queries, &query_selector, alphasort);
    if (n >= 0) {
        int cnt;
        for (cnt = 0; cnt < n; ++cnt) {
            fprintf(cgiOut, "<a href=\"%s%s%s.query=%s\">%s</a>\n", cgiScriptName, cgiPathInfo,
                queries[cnt]->d_name, queries[cnt]->d_name);
        }
    }
```

This code is used in section 88.

99. The *query_selector* above simply rejects filenames that begin with a ‘.’.

```
<Callback procedures 22> +≡
    int query_selector(const struct dirent *entry)
    {
        return ((entry->d_name)[0] ≠ ‘.’);
    }
```

100. To load a requested query, we simply regurgitate the contents of the chosen file.

```
<Load requested query 100> ≡
    strncpy(query_path, query, QUERY_PATH_LEN);
    FILE *query_file = fopen(query_path, "r");
    if (query_file) {
        int c;
        while ((c = fgetc(query_file)) ≠ EOF) {
            <Escape HTML character 101>;
        }
        fclose(query_file);
    }
```

This code is used in section 97.

101. We must write escape all occurrences of ‘<’, ‘>’, and ‘&’ in *applicant.html* for our HTML textarea.

```
<Escape HTML character 101> ≡
    switch (c) {
        case ‘<’: fprintf(cgiOut, "&lt;");
            break;
        case ‘>’: fprintf(cgiOut, "&gt;");
            break;
        case ‘&’: fprintf(cgiOut, "&amp;");
            break;
        default: fputc(c, cgiOut);
    }
```

This code is used in section 100.

102. To save the query, we simply write it to the filename given, provided the filename is not blank. But if the SQL is blank, we interpret this to mean that the query file should be deleted.

```

<Save raw_sql 102> ≡
if (query_name[0] ≡ '\0') {
    fprintf(cgiOut, "Please_type_a_name_by_which_this_query_should_be_known.<br>");
}
else {
    char save_path[QUERY_PATH_LEN];
    strncpy(save_path, query_path, QUERY_PATH_LEN);
    strncat(save_path, query_name, QUERY_PATH_LEN);
    if (raw_sql[0] ≡ '\0') {
        <Unlink save_path 104>;
    }
    else {
        <Write raw_sql to save_path 103>;
    }
}

```

This code is used in section 88.

103. This is a very simple thing to do, but there is always something which may go wrong when file i/o is involved!

```

<Write raw_sql to save_path 103> ≡
FILE *query_file = fopen(save_path, "w");
if (query_file) {
    fprintf(query_file, "%s", raw_sql);
    fclose(query_file);
    fprintf(cgiOut, "<font_color=\"red\">"
        "Your_changes_to_'%s'_have_been_saved!:_:-)"
        "</font><br>\n", query_name);
}
else {
    fprintf(cgiOut, "<font_color=\"red\">"
        "I_could_not_save_to_'%s'!:_:-(<br>"
        "Perhaps_this_is_an_invalid_filename?"
        "</font><br>\n", query_name);
}

```

This code is used in section 102.

104. This is simpler still.

```

<Unlink save_path 104> ≡
unlink(save_path);
fprintf(cgiOut, "<font_color=\"red\">"
    "I_have_removed_the_'%s'_query."
    "</font><br>\n", query_name);

```

This code is used in section 102.

105. A Supplement for CGIC. Here is a procedure we need that CGIC is lacking. If the CGI request is a POST, CGIC unfortunately offers no parsing whatsoever of the query string.

⟨Fundamental procedures and structures 16⟩ +≡

```

int parseQueryString(query_key, buffer, buffer_len)
    char *query_key;
    char *buffer;
    int buffer_len;
{
    char *query_pointer = cgiQueryString;
    while (*query_pointer ≠ '\0') {
        ⟨Copy value if query key matches 106⟩;
        ⟨Find next query key 110⟩;
    }
    return cgiFormNotFound;
}

```

106. If this is the query key we are looking for, copy its value into the buffer.

⟨Copy value if query key matches 106⟩ ≡

```

char *key_pointer = query_key;
do {
    if (*key_pointer ≡ '\0') {
        if (*query_pointer ≡ '=') { /* This is it! We have found the requested query key. */
            ++query_pointer;
            ⟨Copy value of query key 107⟩;
            return cgiFormSuccess;
        }
        else break;
    }
} while (*(key_pointer++) ≡ *(query_pointer++));

```

This code is used in section 105.

107. We copy the value until we reach a NULL or an ampersand.

⟨Copy value of query key 107⟩ ≡

```

while (*query_pointer ≠ '\0' ∧ *query_pointer ≠ '&') {
    ⟨Update buffer_len remaining 108⟩;
    ⟨Copy character 109⟩;
    ++query_pointer;
    ++buffer;
}
*buffer = '\0';

```

This code is used in section 106.

108. If the buffer runs out of room, we end it with a Λ character and return *cgiFormTruncated*;

⟨Update *buffer_len* remaining 108⟩ ≡

```

if (--buffer_len ≡ 0) {
    *buffer = '\0';
    return cgiFormTruncated;
}

```

This code is used in section 107.

109. All occurrences of '+' must be changed to ' ' and all occurrences of '%' followed by two hexadecimal digits must be changed to the character whose ascii code is the hexadecimal number given.

```

<Copy character 109> ≡
switch (*query_pointer) {
case '%':
    {
        <Convert hexadecimal to ascii character 111>;
        break;
    }
case '+':
    {
        *buffer = '␣';
        break;
    }
default:
    {
        *buffer = *query_pointer;
    }
}

```

This code is used in section 107.

110. When we need to find the next query key, we simply advance until we reach an ampersand, then we pass it.

```

<Find next query key 110> ≡
while (*query_pointer ≠ '&') {
    if (*query_pointer ≡ '\0') return cgiFormNotFound;
    ++query_pointer;
}
++query_pointer;

```

This code is used in section 105.

111. Now all we have left to implement is the hexadecimal conversion to ascii.

```

<Convert hexadecimal to ascii character 111> ≡
int ascii = 0;
char digit;
digit = *(++query_pointer);
<Add hexadecimal digit 112>;
ascii <<= 4; /* The first digit is the high digit */
digit = *(++query_pointer);
<Add hexadecimal digit 112>;
*buffer = (char) ascii;

```

This code is used in section 109.

112. If anything is wrong with a hexadecimal digit, then we will return an error code.

⟨Add hexadecimal digit 112⟩ ≡

```
if (digit < '0' ∨ digit > 'f') return cgiFormBadType;  
if (digit ≤ '9') {  
    ascii += (int)(digit - '0');  
}  
else {  
    if (digit > 'F') digit -= 32;    /* Convert to upper case */  
    if (digit < 'A' ∨ digit > 'F') return cgiFormBadType;  
    ascii += (int)(digit - 'A' + 10);  
}
```

This code is used in section 111.

113. HTML Markup.

114. Here follows the pretty graphical stuff.

115. We have chosen good old fashioned HTML.

```
<Set content type 115> ≡
    cgiHeaderContentType("text/html");
```

This code is used in section 5.

116. So let us begin!

```
<Begin HTML document 116> ≡
    fprintf(cgiOut, "<html>\n");
```

This code is used in section 5.

117. We will use a horizontal rule to divide disparate material.

```
<Display HTML horizontal rule 117> ≡
    fprintf(cgiOut, "<hr>\n");
```

This code is used in sections 10 and 88.

118. A bordered HTML table is a thing of simple beauty.

```
<Begin HTML table 118> ≡
    fprintf(cgiOut, "<table border>\n");
```

This code is used in sections 23, 26, 50, 53, and 94.

119. Each row begins thus:

```
<Begin table row 119> ≡
    fprintf(cgiOut, "<tr>\n");
```

This code is used in sections 23, 29, 31, 55, 59, 60, and 61.

120. or thus:

```
<Begin highlit table row 120> ≡
    fprintf(cgiOut, "<tr bgcolor=\"yellow\">");
```

This code is used in section 61.

121. The header is bolded. Let's give it some color too!

```
<Begin table header 121> ≡
    fprintf(cgiOut, "<th bgcolor=\"lightblue\">");
```

This code is used in sections 23, 29, and 55.

122. And ends thus.

```
<End table header 122> ≡
    fprintf(cgiOut, "</th>\n");
```

This code is used in sections 23, 29, and 55.

123. Each cell begins thus:

```
<Begin table cell 123> ≡
    fprintf(cgiOut, "<td>");
```

This code is used in sections 23, 31, 55, 59, 60, and 61.

124. May contain a NULL value,

```
<Display HTML NULL 124> ≡
    fprintf (cgiOut, "&nbsp");
```

This code is used in section 32.

125. And likewise ends.

```
<End table cell 125> ≡
    fprintf (cgiOut, "</td>");
```

This code is used in sections 23, 31, 55, 59, and 60.

126. The row ends.

```
<End table row 126> ≡
    fprintf (cgiOut, "</tr>\n");
```

This code is used in sections 23, 29, 31, 55, and 60.

127. Indeed, all good things must come to an end.

```
<End HTML table 127> ≡
    fprintf (cgiOut, "</table>\n");
```

This code is used in sections 23, 26, 50, 53, and 94.

128. And now for the grand finale! Take a bow, maestro.

```
<End HTML document 128> ≡
    fprintf (cgiOut, "</html>\n");
```

This code is used in sections 4, 5, and 14.

action: 88, [89](#).

ACTION_LEN: [88](#), [89](#).

alphasort: 98.

APPEND_TO_FROM: [41](#), [46](#).

APPEND_TO_INSERT: [66](#), [72](#), [79](#).

APPEND_TO_SELECT: [40](#), [45](#), [48](#), [49](#).

APPEND_TO_UPDATE: [65](#), [71](#), [78](#), [80](#).

APPEND_TO_VALUES: [66](#), [72](#), [79](#), [81](#).

APPEND_TO_WHERE: [42](#).

applicant: 101.

ARG: 3.

argc: [22](#), [28](#), [29](#), [31](#), [37](#), [52](#), [55](#), [86](#), [96](#).

argv: [22](#), [28](#), [31](#), [32](#), [37](#), [52](#), [57](#), [59](#), [61](#), [63](#), [86](#), [96](#).

ascii: [111](#), [112](#).

buffer: [105](#), [107](#), [108](#), [109](#), [111](#).

buffer_append: [33](#), [40](#), [41](#), [42](#), [65](#), [66](#), [82](#).

buffer_append_with_apostrophies: 80, [81](#), [82](#).

BUFFER_END: [33](#), [82](#).

buffer_end_ref: [33](#), [82](#).

buffer_len: [105](#), [108](#).

c: [100](#).

CALLBACK: 3.

cgi_result: [6](#), [64](#), [80](#), [81](#).

cgiCookieString: 6.

cgiFormBadType: 112.

cgiFormEmpty: 64, [80](#), [81](#).

cgiFormEntries: 73, [74](#).

cgiFormNotFound: 105, [110](#).

cgiFormString: 6, [64](#), [80](#), [81](#), [89](#).

cgiFormSuccess: 6, [9](#), [10](#), [42](#), [64](#), [73](#), [74](#), [80](#), [81](#), [106](#).

cgiFormTruncated: 108.

cgiHeaderContentType: 115.

cgiHeaderCookieSetString: 7.

cgiHtmlEscape: 30, [32](#), [55](#), [57](#), [91](#), [92](#), [96](#), [97](#).

cgiMain: [5](#).

cgiOut: 4, [8](#), [11](#), [13](#), [14](#), [23](#), [24](#), [25](#), [32](#), [33](#), [52](#), [53](#), [56](#), [57](#), [59](#), [60](#), [61](#), [70](#), [77](#), [82](#), [87](#), [90](#), [91](#), [92](#), [95](#), [97](#), [98](#), [101](#), [102](#), [103](#), [104](#), [116](#), [117](#), [118](#), [119](#), [120](#), [121](#), [122](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#).

cgiPathInfo: 8, [11](#), [13](#), [23](#), [24](#), [25](#), [53](#), [59](#), [60](#), [87](#), [90](#), [98](#).

cgiQueryString: 23, [50](#), [53](#), [54](#), [59](#), [60](#), [69](#), [105](#).

cgiScriptName: 7, [8](#), [11](#), [23](#), [24](#), [25](#), [53](#), [59](#), [60](#), [87](#), [90](#), [98](#).

cgiServerName: 7.

cgiStringArrayFree: 73, [74](#).

cnt: [98](#).

columnNames: [22](#), [28](#), [30](#), [32](#), [37](#), [52](#), [55](#), [57](#), [63](#), [86](#), [87](#), [96](#).

compile_tables_list: 21, [22](#).

copy_string_to_list: [18](#), [22](#), [38](#), [44](#).

create_sql_for_table: 26, [44](#), 50.
d_name: 98, 99.
database: [3](#), 14, 15, 67.
 DESTINATION: 18.
digit: [111](#), 112.
dirent: 2, 98, 99.
display_column_names: 26, [27](#), 29, 50, 94.
 DUPLICATE_STRING: 11, [18](#), 50, 63, 87.
effective_query: 11, 67, [68](#), 69, 87.
end: [67](#).
entry: [99](#).
 EOF: 100.
equal_pointer: [50](#).
error_code: [3](#), 4.
error_text: [3](#), 4, 14.
exit: 4, 13, 14, 33, 77, 82.
fclose: 100, 103.
fgetc: 100.
field: [37](#), 38, 40, 41, 42, [75](#), [76](#), 78, 79.
field_number: [49](#).
filepath: [9](#), 13, 14, 88.
fopen: 100, 103.
 FOREIGN_TABLE: [87](#).
form_entries: [73](#), [74](#), 75, 76.
form_entry: [75](#), [76](#), 80, 81.
fprintf: 4, 8, 11, 13, 14, 23, 24, 25, 32, 33, 52, 53, 56, 57, 59, 60, 61, 70, 77, 82, 87, 90, 91, 92, 95, 97, 98, 101, 102, 103, 104, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128.
fputc: 101.
free: 17, 50, 63, 87.
free_list: [17](#), 44.
from_buffer: 41, [44](#), 46, 48.
from_buffer_end: [34](#), 41, 46.
 FROM_BUFFER_LENGTH: [44](#), 46.
from_buffer_length: [34](#), 41, 46.
head: [18](#).
html: 101.
i: [29](#), [31](#), [37](#), [55](#), [67](#), [75](#), [76](#), [96](#).
insert_buffer: [66](#).
insert_buffer_end: [66](#), 72.
 INSERT_BUFFER_LEN: [66](#).
insert_buffer_length: [66](#).
 INSERT_REQUESTED: [52](#), 84.
insert_value: [81](#).
 INSERT_VALUE_LEN: [81](#).
insert_values: [66](#), 72.
insert_values_end: [66](#), 81.
 INSERT_VALUES_LEN: [66](#).
insert_values_length: [66](#), 81.
interactive: 26, [27](#), 31, 50.
iterator: [23](#).
j: [67](#).
join_tables: [35](#), [36](#), 38, 44, 45.
join_tables_callback: 35, [37](#).
key: 51.
 KEY_LEN: 10, [12](#).
key_pointer: [106](#).
last_ampersand: [11](#).
last_equal_sign: [87](#).
 LENGTH_REMAINING: [33](#), [82](#).
length_remaining_ref: 33, 82.
list: [16](#), 17, 18, 19, 20, 23, 34, 83.
list_pointer: [18](#).
lst: [17](#), [20](#).
malloc: 18, 69.
n: [98](#).
name: [32](#).
 NEW_INSTANCE: [18](#).
new_query_string: [11](#), [53](#), 54, [87](#).
next: [16](#), [17](#), 18, 20, 23, 83.
offset: [67](#).
parameter: [89](#), 91, 93.
 PARAMETER_LEN: [88](#), 89.
parameter_sql: [88](#), 93, 94, 95.
 PARAMETER_SQL_LEN: [88](#), 93.
pArg: [22](#), [28](#), 30, [37](#), 40, 41, [52](#), 56, 59, 61, [86](#), [96](#).
parseQueryString: 9, 10, 42, [105](#).
 PASSWORD: [6](#), 7.
password_given: [6](#).
password_is_correct: [5](#), 6.
 PASSWORD_LEN: [6](#).
 PATH_INFO: 9.
primary_key: 10, [12](#), 51, 52, 56, 67.
queries: [98](#).
query: [9](#), 92, 97, 100.
query_file: [100](#), [103](#).
query_key: [105](#), 106.
 QUERY_LEN: [9](#).
query_name: [89](#), 92, 102, 103, 104.
 QUERY_NAME_LEN: [88](#), 89.
query_path: [88](#), 98, 100, 102.
 QUERY_PATH_LEN: 88, [98](#), 100, 102.
query_pointer: [50](#), [105](#), 106, 107, 109, 110, 111.
query_selector: 98, [99](#).
query_string_copy: [50](#).
raw_sql: [89](#), 93, 97, 102, 103.
 RAW_SQL_LEN: [88](#), 89.
save_path: [102](#), 103, 104.
scandir: 98.
search_fields: [85](#).
search_fields_callback: 85, [86](#).
 SEARCH_FIELDS_LEN: [85](#).
select_buffer_end: [34](#), 40, 45.

select_buffer_length: [34](#), 40, 45.
select_field_count: [39](#), 40, 45, 49.
select_first_radio: 61, [62](#), 63.
select_row: [51](#).
SELECT_ROW_LEN: [51](#).
selected_radio_key: 42, 48, [58](#), 59, 60, 61, 63.
show_fields: [35](#).
SHOW_FIELDS_LEN: [35](#).
SHOW_TABLES: [21](#).
skip_redundant_fields: 42, [43](#), 44.
sprintf: 35, 51, 64, 85, 93.
SOURCE: 18.
sprintf: 67.
SQL: 3.
sql_buffer: [26](#), [44](#), 45.
SQL_BUFFER_LENGTH: [26](#), 50.
sql_buffer_length: [44](#), 45.
sql_context_buffer: [50](#).
sqlite: 3.
SQLITE_BUSY: 14.
sqlite_busy_timeout: 14.
sqlite_close: 15.
SQLITE_EXEC: [3](#), 14, 21, 26, 35, 50, 51, 65, 66, 85, 94, 95.
sqlite_exec: 3.
sqlite_last_insert_rowid: 67.
SQLITE_OK: 4.
sqlite_open: 3, 14.
start: [67](#).
strchr: 50, 54, 67.
strcmp: 6, 20, 61, 75, 76, 83, 86, 88.
strcpy: 18, 69.
STRING: 40, 41, 42, 65, 66.
string: [16](#), 17, [18](#), [20](#), 23, [33](#), 38, 44, [82](#), 83, 85.
string_exists: [20](#), 38, 50, 55.
string_remaining: [33](#), [82](#).
strlen: 18, 30, 67, 69.
strncat: 88, 100, 102.
strncmp: 30.
strncpy: 88, 102.
strrchr: 11, 87.
table: [9](#), 10, 26, [35](#), [36](#), [44](#), 45, 46, 48, 51, 60, [63](#), 64, 71, 72, 75, 76, 83, 85, 87.
table_browser_callback: 26, [28](#), 50, 94.
table_iterator: [83](#), 85.
TABLE_LEN: [9](#), 10.
table_name_len: [30](#).
TABLE_VAR: [9](#), 10, 11, 23.
tables: [19](#), 22, 23, 38, 50, 55, 83.
TYPE: 18.
unformatted_callback: 95, [96](#).
unlink: 104.
update_buffer: [65](#).
update_buffer_end: [65](#), 71, 80.
UPDATE_BUFFER_LEN: [65](#).
update_buffer_length: [65](#), 80.
update_key_name: [64](#).
UPDATE_KEY_NAME_LEN: [64](#).
update_key_value: [64](#), 71.
UPDATE_KEY_VALUE_LEN: [64](#).
update_row_callback: 51, [52](#).
update_value: [80](#).
UPDATE_VALUE_LEN: [80](#).
value: [42](#).
VALUE_LEN: [42](#).
visited: [34](#), 38, 44.
where_buffer: [44](#), 47, 48.
where_buffer_end: [34](#), 42, 47.
WHERE_BUFFER_LEN: [34](#).
WHERE_BUFFER_LENGTH: [44](#), 47.
where_buffer_length: [34](#), 42, 47.

- <Abandon attempt to process update request 77> Used in sections 73, 74, 80, and 81.
- <Add field and value to INSERT statement 79> Used in section 76.
- <Add field and value to UPDATE statement 78> Used in section 75.
- <Add hexadecimal digit 112> Used in section 111.
- <Append ORDER BY clause to *sql_buffer* 49> Used in section 44.
- <Append field to SELECT clause 40> Used in section 38.
- <Append fields to insert 74> Used in section 72.
- <Append fields to update 73> Used in section 71.
- <Append join to *from_buffer_end* 41> Used in section 38.
- <Append restriction to *where_buffer_end* 42> Used in section 38.
- <Append value to INSERT statement 81> Used in section 79.
- <Append value to UPDATE statement 80> Used in section 78.
- <Append *from_buffer* and *where_buffer* to *sql_buffer* 48> Used in section 44.
- <Ask for password 8> Used in section 5.
- <Begin HTML document 116> Used in section 5.
- <Begin HTML table 118> Used in sections 23, 26, 50, 53, and 94.
- <Begin highlit table row 120> Used in section 61.
- <Begin row with primary key 59> Used in section 31.
- <Begin row with radio button 61> Used in section 59.
- <Begin table cell 123> Used in sections 23, 31, 55, 59, 60, and 61.
- <Begin table header 121> Used in sections 23, 29, and 55.
- <Begin table row 119> Used in sections 23, 29, 31, 55, 59, 60, and 61.
- <Browse table 26> Used in sections 10 and 63.
- <Build SQL INSERT statement 72> Used in section 66.
- <Build SQL UPDATE statement 71> Used in section 65.
- <Callback procedures 22, 28, 37, 52, 86, 96, 99> Used in section 1.
- <Check password 6> Used in section 5.
- <Close database 15> Used in sections 4 and 9.
- <Convert hexadecimal to ascii character 111> Used in section 109.
- <Copy character 109> Used in section 107.
- <Copy value if query key matches 106> Used in section 105.
- <Copy value of query key 107> Used in section 106.
- <Copy *cgiQueryString* 69> Used in section 10.
- <Create radio buttons 63> Used in section 55.
- <Create text entry 57> Used in section 55.
- <Create update form 53> Used in section 52.
- <Dispatch field for SQL creation 38> Used in section 37.
- <Display HTML NULL 124> Used in section 32.
- <Display HTML horizontal rule 117> Used in sections 10 and 88.
- <Display SQL form input 97> Used in section 90.
- <Display 'home' link 25> Used in section 9.
- <Display 'query' link 24> Used in section 9.
- <Display 'tables' link 11> Used in section 10.
- <Display column names if we have not yet 29> Used in section 28.
- <Display drilldown links 83> Used in section 84.
- <Display form for direct SQL access 90> Used in section 88.
- <Display link for new row 60> Used in section 26.
- <Display link to foreign table 87> Used in section 86.
- <Display parameter input 91> Used in section 90.
- <Display query context 50> Used in section 10.
- <Display saved query filename input 92> Used in section 90.
- <Display saved query links 98> Used in section 88.

<Display tables list 23> Used in section 10.
<Display the primary key for the update form 56> Used in section 55.
<Display update form 51> Used in section 10.
<Display *argv*[*i*] 32> Used in section 31.
<Display *argv* array 31> Used in section 28.
<Display *columnNames*[*i*] 30> Used in section 29.
<End HTML document 128> Used in sections 4, 5, and 14.
<End HTML table 127> Used in sections 23, 26, 50, 53, and 94.
<End table cell 125> Used in sections 23, 31, 55, 59, and 60.
<End table header 122> Used in sections 23, 29, and 55.
<End table row 126> Used in sections 23, 29, 31, 55, and 60.
<Escape HTML character 101> Used in section 100.
<Execute *parameter_sql* and write HTML table 94> Used in section 88.
<Execute *parameter_sql* and write unformatted 95> Used in section 88.
<Find next query key 110> Used in section 105.
<Fundamental procedures and structures 16, 17, 18, 33, 36, 44, 82, 105> Used in section 1.
<Get required filepath 13> Used in section 9.
<Give up if there was any SQL error 4> Used in sections 14, 21, 35, 51, 65, 66, 85, 94, and 95.
<Global variables 3, 12, 19, 27, 34, 39, 43, 58, 62, 68> Used in section 1.
<If necessary, perform updates 64> Used in section 10.
<Included files 2> Used in section 1.
<Initialize *from_buffer* 46> Used in section 44.
<Initialize *sql_buffer* 45> Used in section 44.
<Initialize *where_buffer* 47> Used in section 44.
<Iterate through form variables for INSERT 76> Used in section 74.
<Iterate through form variables for UPDATE 75> Used in section 73.
<Load requested query 100> Used in section 97.
<Locate *new_query_string* 54> Used in section 53.
<Main CGI method 5> Used in section 1.
<Make form elements for fields 55> Used in section 53.
<Open or create database 14> Used in section 9.
<Parameterize sql statement 93> Used in section 88.
<Perform insert 66> Used in section 64.
<Perform updates 65> Used in section 64.
<Populate tables list 21> Used in section 10.
<Pretend that we are updating the new row 67> Used in section 66.
<Procedures 20, 35> Used in section 1.
<Provide abstracted SQL access 10> Used in section 9.
<Provide direct SQL access 88> Used in section 9.
<Read direct SQL form variables 89> Used in section 88.
<Report successful update 70> Used in sections 65 and 66.
<Respond to valid user 9> Used in section 5.
<Save *raw_sql* 102> Used in section 88.
<Search fields for foreign key for our table 85> Used in section 83.
<Set content type 115> Used in section 5.
<Set password cookie 7> Used in section 5.
<Unless form is for INSERT, display drilldown links 84> Used in section 10.
<Unlink *save_path* 104> Used in section 102.
<Update *buffer_len* remaining 108> Used in section 107.
<Write *raw_sql* to *save_path* 103> Used in section 102.

CGISQLITE

	Section	Page
Table Browsing	26	9
Displaying the Query Context	50	17
Displaying an Update Form	51	18
Displaying the Drilldown Links	83	28
Direct SQL access	88	30
A Supplement for CGIC	105	35
HTML Markup	113	38