The Psql Correspondence back-end

(Work in progress—handle with care)
$Id: libpsql.w,v 1.28 2006/11/15 22:14:49 tlaronde Exp

Thierry LARONDE

Abstract

Here is a `psql` back-end for the KerGIS `libcorr`.
For every session, a |transaction| with a PostgreSQL server is started using the `psql` client.
Communication with the `psql` client is done via two pipes.
The transaction (one transaction per connection) is only committed if everything went fine.

Une implémentation d'un interfaçage avec `psql` pour la `libcorr`.
Chaque ouverture d'une session provoque la création d'une connexion avec un serveur PostgreSQL via le client `psql`. La communication avec `psql` se fait par deux pipes, et la session est une transaction. Si tout s'est correctement déroulé, à la fermeture les éventuelles modifications sont réalisées (SQL `commit`). Sinon, elles sont annulées.

November 26, 2006 at 11:44

1. Licence.

2. Public interface to the library.

We are going to describe the visible top of the iceberg first: what callers are supposed to see and use, callers being here routines in the core `libcorr` library and not userland programs.

The naming conventions are the KerGIS ones. The API conventions are the KerGIS ones too: useful values are returned by putting the result in a memory place, the returning value being whether **void**, if no error is significant, or an **int** specifying the status: `G_SUCCESS` $\equiv 0$ for OK, or the error number if there was one.

⟨ `corr/psql.h` 2 ⟩ ≡
```
#ifndef C_PSQL_H
#define C_PSQL_H
#include "gis.h"
#include "corr.h"
  ⟨ Public prototypes 4 ⟩
#endif    /* C_PSQL_H */
```

3. Defaults.
Some values are mandatory, and if not set some defaults are used. You should better know what they are!

```
⟨ Check crucial values 3 ⟩ ≡
  if (Psql→Head→dbase ≡ Λ ∨ Psql→Head→table ≡ Λ) {
    status ← C_psql_EMISSINGVALUES;
    goto end;
  }
  if (Psql→Head→datum_group ≡ Λ ∧ Psql→Head→datum_group_name ≡ Λ) {
    status ← C_psql_EMISSINGSELECTION;
    goto end;
  }
  if (Psql→Head→separator ≡ '\0') Psql→Head→separator ← '|';
  if (Psql→Head→null_as ≡ Λ) Psql→Head→null_as ← "NULL";
  if (Psql→Head→encoding ≡ Λ) Psql→Head→encoding ← KT_ICONV_UTF_8;
```

This code is used in section 14.

4. Opening/closing a connexion.
The connection to the postmaster will be done via `psql`. The informations are passed via the *C_DFile* structure.

⟨ Public prototypes 4 ⟩ ≡
int *C_psql_open*(**struct** *C_DFile* ∗*Psql*);

See also sections 5, 6, and 7.

This code is used in section 2.

5. Closing, from the caller's standpoint is quite simple.

⟨ Public prototypes 4 ⟩ +≡
int *C_psql_close*(**struct** *C_DFile* ∗*Psql*);

6. Querying attributes associated with a group id.

An attribute is a record in the selected table, identified by its *oid*. We set *Att→id* ← *oid*, we will retrieve all the records associated with a group id (the image).

For efficiency purposes, we try to retrieve all the attributes in one command transmitted to the *postmaster* via the *psql* client. But for memory preservation, we will return only a maximum of *chunk_size* attributes on every call. Hence, if *multiplicity* > *chunk_size*, we cache the attributes in a file that will be read on further invocations and that will be invalidated if the *offset* requested does not match the current *image_offset* in it.

⟨ Public prototypes 4 ⟩ +≡

int *C_psql_get_image*(**struct** *C_DFile* ∗*Psql*, **struct** *C_Image* ∗*Image*);

7. Manipulating the records.

When modifying a record, the caller must call the function *C_psql_set_image*. Mode is set by flags in the *Att* structure.

If modifying an att, it will write it at the specified address. If the record is new, it is appended.

⟨ Public prototypes 4 ⟩ +≡

int *C_psql_set_image*(**struct** *C_DFile* ∗*Psql*, **struct** *C_Image* ∗*Image*);

8. Implementation.

The details of the manipulations are hidden from the public. Furthermore, critical data structures are kept strictly internal to avoid the mess.

The naming conventions are the ones adopted for the whole KerGIS sources.

```
⟨ psql/_psql.h 8 ⟩ ≡
#ifndef _C_PSQL_H
#define _C_PSQL_H
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>     /* system */
#include <string.h>
#include <unistd.h>     /* unlink */
#include <sys/types.h>      /* off_t */
#include "ksys/cdefs.h"
#include "ksys/endian.h"      /* portability stuff */
#include "gis.h"
#include "corr.h"
#include "corr/psql.h"
  ⟨ Private macros 18 ⟩
  ⟨ Private structures 16 ⟩
  ⟨ Private prototypes 52 ⟩
#endif      /* _C_PSQL_H */
```

9. Communication with the coprocess.

The most important is to set the connection with our coprocess: `psql`. We will not use a pseudo terminal, since `psql` can do without and separating commands from result is far better. Furthermore, when trying to read from a pty, we will have to take into account the window size and so on, and to develop at least some scanner to concat lines that break at window's edge.

So communication with two pipes will be good enough.

```
⟨ Start our psql coprocess 9 ⟩ ≡
  {
    ⟨ Set signal handling 51 ⟩
    if ((_PID ← fork( )) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if (_PID) {      /* parent */
      (void) close(_ADMIN→fds1[0]);      /* close read1 */
      (void) close(_ADMIN→fds2[1]);      /* close write2 */
      if ((_CMD ← fdopen(_ADMIN→fds1[1], "a")) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      if (setvbuf(_CMD, Λ, _IOLBF, (size_t) 0)) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      if ((_RESULT ← fdopen(_ADMIN→fds2[0], "r")) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
    }
    else {      /* in the child */
      char *arg_null;
      char arg_separator[2];
      ⟨ Manage child pipes 10 ⟩
      if ((arg_null ← (char *) malloc(strlen(Psql→Head→null_as) + 8)) ≡ Λ)
        G_msg(errno ≪ G_ERROR_SHIFT, G_Errors, G_FATAL);
      (void) sprintf(arg_null, "null=%s", Psql→Head→null_as);
      (void) sprintf(arg_separator, "%c", Psql→Head→separator);
      if (execlp("psql", "psql", "-AtqF", arg_separator, "-P", arg_null, "-v", "ON_ERROR_STOP=YES",
            Psql→Head→dbase, (char *) 0) < 0)  G_msg(errno ≪ G_ERROR_SHIFT, G_Errors, G_FATAL);
    }
  }
```

This code is used in section 14.

10. The child pipes are the converse of the parent, and we associate them to stdin and stdout.

```
⟨ Manage child pipes 10 ⟩ ≡
  (void) close(_ADMIN→fds1[1]);      /* close write1 */
  (void) close(_ADMIN→fds2[0]);      /* close read2 */
  if (_ADMIN→fds1[0] ≠ STDIN_FILENO) {
    if (dup2(_ADMIN→fds1[0], STDIN_FILENO) ≠ STDIN_FILENO)
      G_msg(errno ≪ G_ERROR_SHIFT, G_Errors, G_FATAL);
    (void) close(_ADMIN→fds1[0]);
  }
  if (_ADMIN→fds2[1] ≠ STDOUT_FILENO) {
    if (dup2(_ADMIN→fds2[1], STDOUT_FILENO) ≠ STDOUT_FILENO)
      G_msg(errno ≪ G_ERROR_SHIFT, G_Errors, G_FATAL);
    (void) close(_ADMIN→fds2[1]);
  }
```

This code is used in section 9.

11. The problem that occurs now is to be able to interpret without a complex scanner the result sent by the client.

We have seen that, even in not aligned output without comments and so on, sometimes leading blank lines are there, and sometimes not (there may be a blunder about the way we scan the result leaving a newline, but I didn't find it at the moment). So when we are sure there must be something to read (to simply avoid sleeping when the client waits after us to send commands...), we will skip blank lines.

G_fread_line is used, and *buf* is declared in every function.

```
⟨ Skip blank lines to usefull data in fp_in 11 ⟩ ≡
  for (G_fread_line(&buf, fp_in); buf ≠ Λ ∧ ¬*buf; ) {
    free(buf);
    G_fread_line(&buf, fp_in);
  }
```

This code is used in sections 21, 34, 36, and 38.

12. The session is *transaction* based. If everything went fine, it is committed. If not, rollback is requested. We track the status with the `C_INVALID_TRANSACTION` flag.

To avoid buffering on the *psql* side (this is not interactive use, and this slow down things considerably) we unset the *psqlpager*.

```
⟨ Begin transaction 12 ⟩ ≡
  (void) fprintf(_CMD, "BEGIN;\n");
```

This code is used in section 14.

13. Committing or cancelling depending on problems tracked with `C_INVALID_TRANSACTION`.

```
⟨ Commit transaction if ¬C_INVALID_TRANSACTION; else rollback 13 ⟩ ≡
  if (¬(Psql→flags & C_INVALID_TRANSACTION)) (void) fprintf(_CMD, "COMMIT;\n");
  else (void) fprintf(_CMD, "ROLLBACK;\n");
```

This code is used in section 22.

14. Opening and closing.
Depending on the *mode*, the table is created or simply open.

```
⟨ psql/open.c 14 ⟩ ≡
#include <termios.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "psql/_psql.h"
  int C_psql_open(struct C_DFile *Psql)
  {
    int status ← G_SUCCESS;      /* return value */
    unsigned long i;
    char *buf;      /* set by G_fread_line */
    FILE *fp_in;      /* from where we get data */
    Psql→handler ← −1;
    ⟨ Check crucial values 3 ⟩
    ⟨ Allocate and init new _psql_Proc; return error if failed 19 ⟩
    ⟨ Start our psql coprocess 9 ⟩
    ⟨ Begin transaction 12 ⟩
    ⟨ Get or set table definition and nb_atts 20 ⟩
    ⟨ Set table_desc 47 ⟩
    ⟨ Init the selection structures 26 ⟩
  end:
    if (status) {
      Psql→flags |= C_INVALID_TRANSACTION;
      (void) C_psql_close(Psql);
    }
    return status;
  }
```

15. Some values are mandatory: the database and the table.

16. The internal handling is done via an internal structure *_psql_Proc*, that is accessed according to the *handler* set in the *Psql* struct that will be used on every call of the library.

At the moment, multiplexing is not handled: if there is a broken pipe for the process (whether caused by "us" or in the caller program, we mark the handler as dead). To multiplex will need more thinking (since signals are process based, and we need to deal with action done by the calling program).

```
⟨ Private structures 16 ⟩ ≡
  struct _psql_Proc {
    int pid;      /* track child */
    int fds1[2];
    int fds2[2];
    FILE *cmd;
    FILE *result;
    char *table_desc;
    ⟨ Selection structures 24 ⟩
  };
#ifdef HIC
#define EXTERN
#else
#define EXTERN  extern
#endif      /* we define a maximum number of group of files to handle */
#define _MAX_HANDLERS  8      /* minimum value of OPEN_MAX / 2 */
  EXTERN
      int nb_psql_handlers;      /* nb of handlers actually in use */
      EXTERN
          struct _psql_Proc *psql_Handlers[_MAX_HANDLERS]; EXTERN
              struct sigaction oact_pipe;      /* keep previous signal config */
#define _ADMIN  (psql_Handlers[Psql→handler])
#define _PID_ADMIN→pid
#define _CMD_ADMIN→cmd
#define _RESULT_ADMIN→result
#define _DATUMS(Psql→Data)→Datums
```

This code is used in section 8.

17. HIC will be defined in an object that needs to be here at last, that is in the closing routine object file.

18. Since we need to ensure that the *handler* set is a correct one, we define a macro.

Pipes could have been closed. At the moment, we do not try to identify the responsible for this and we simply mark **all** the handlers as invalid, not by freeing them (there are clients that still think these numbers are valid and painfull reality will only be said to them on next call), but we set _PID $\leftarrow 0$ and release resources.

So _PID $\equiv 0$ is one of the test to be made when called to see if the handler is valid or is still valid.

```
⟨ Private macros 18 ⟩ ≡
#define _CHECK_HANDLER
  if (Psql→handler < 0 ∨ Psql→handler > _MAX_HANDLERS ∨ _ADMIN ≡ Λ ∨ _PID ≡ 0) return C_EBADHANDLER;
```

See also section 25.

This code is used in section 8.

19. If there is no handler left, we can bail out. After having allocated the structure, we will be able to use *C_psql_close*. Before that, we do not go to end on error, but return (with error) from where we are.

We use *calloc* to ensure everything is zero, and simply update defaults that should not have this value.

```
⟨ Allocate and init new _psql_Proc; return error if failed 19 ⟩ ≡
  if (nb_psql_handlers ≥ _MAX_HANDLERS) return C_EOPENMAX;
  for (Psql→handler ← 0; Psql→handler < _MAX_HANDLERS ∧ psql_Handlers[Psql→handler] ≠ Λ; ++Psql→handler)
    ;
  if ((psql_Handlers[Psql→handler] ← (struct _psql_Proc *) calloc(1, sizeof(struct _psql_Proc))) ≡ Λ) {
    Psql→handler ← −1;
    return errno ≪ G_ERROR_SHIFT;
  }
  _ADMIN→fds1[0] ← −1;
  _ADMIN→fds1[1] ← −1;
  _ADMIN→fds2[0] ← −1;
  _ADMIN→fds2[1] ← −1;
  ++nb_psql_handlers;
  if (pipe(_ADMIN→fds1) < 0 ∨ pipe(_ADMIN→fds2) < 0) {
    status ← errno ≪ G_ERROR_SHIFT;
    goto end;
  }
```

This code is used in section 14.

20. On opening, we retrieve the informations about the table from the server, or we set if in creation mode.

```
⟨ Get or set table definition and nb_atts 20 ⟩ ≡
  fp_in ← _RESULT;      /* we use our stdin */
  if (Psql→mode ≡ G_O_WRONLY) {
    ⟨ Convert Data to SQL table 41 ⟩
    Psql→Data→nb_atts ← 0;
  }
  else {
    Psql→Data→Datums ← Λ;      /* will be read from server */
    ⟨ Convert SQL table to Data 43 ⟩
    ⟨ Get nb_atts 21 ⟩
  }
```

This code is used in section 14.

21. Getting the number of atts is just a matter of passing the correct command.

If there is a problem when requesting for this (access permissions) we can bail out.

```
⟨ Get nb_atts 21 ⟩ ≡
  (void) fprintf(_CMD, "SELECT␣count(*)␣FROM␣%s;\n", Psql→Head→table);
  ⟨ Skip blank lines to usefull data in fp_in 11 ⟩
  if (buf ≡ Λ) {
    status ← C_psql_ECMD;
    goto end;
  }
  (void) sscanf(buf, "␣%lu␣", &Psql→Data→nb_atts);
  free(buf);
```

This code is used in section 20.

22. Closing: *C_psql_close*.

Closing is fairly obvious. Some supplementary checks shall be done and will be added in the future.

```
⟨ psql/close.c 22 ⟩ ≡
#include <unistd.h>
#define HIC
#include "psql/_psql.h"
  int C_psql_close(struct C_DFile *Psql)
  {
    int status ← G_SUCCESS;
    int cache;      /* be able to be safely called when not opened (interrupted user program */
    if (Psql→handler ≡ −1) return G_SUCCESS;
    _CHECK_HANDLER;

    ⟨ Commit transaction if ¬C_INVALID_TRANSACTION; else rollback 13 ⟩
    if (_CMD ≠ Λ) (void) fclose(_CMD);
    if (_RESULT ≠ Λ) (void) fclose(_RESULT);
    if (_ADMIN→fds1[1] ≥ 0) (void) close(_ADMIN→fds1[1]);
    if (_ADMIN→fds2[0] ≥ 0) (void) close(_ADMIN→fds2[0]);
    free(_SELECT_FORMAT);
    free(_ADMIN→table_desc);
    for (cache ← 0; cache < _NB_CACHES; cache++)
      if (_CACHES[cache].fp ≠ Λ) ⟨ Invalidate the cache entry 39 ⟩
    free(psql_Handlers[Psql→handler]);
    psql_Handlers[Psql→handler] ← Λ;
    Psql→handler ← −1;
    −−nb_psql_handlers;
    return status;
  }
```

23. Getting an image.

The first step is to select the attributes (to build the *select_clause*).

Depending on the value of the *group* specified in the *Image* structure, we take into account the special cases (P_GROUP_EMPTY, P_GROUP_ALL) to map them to the corresponding values in the back-end dataset manager.

Once this is done, we can query the *multiplicity*. The decision about allocating a *Cache* (i.e. a file storing the whole attributes associated to a group) or not is taken depending on the multiplicity. It it is less than *chunk_size*, we proceed "on fly" taking everything from the pipe. If not, we store in a cache that will be read on next calls.

24. Since the selection clause is fixed at open time (for the purpose of the KerGIS correspondence), we can compute it once. It will be a simple format for `sprintf`.

```
⟨ Selection structures 24 ⟩ ≡
  char *select_format;
#define _SELECT_FORMAT_ADMIN→select_format
```

See also section 28.

This code is used in section 16.

25. Depending on whether we might use the group (number) or the group name, things are slightly different, since we have to do quoting in the string case. But the building of the *select_format* is just what we do do by hand.

```
⟨ Private macros 18 ⟩ +≡
#define _USE_GROUP_NAME (°2 ≪ C_FLAGS_SHIFT)
```

26.

```
⟨ Init the selection structures 26 ⟩ ≡
  {
    char *column;
    if (Psql→Head→datum_group ≡ Λ) Psql→flags |= _USE_GROUP_NAME;
    column ← Psql→flags & _USE_GROUP_NAME ? Psql→Head→datum_group_name : Psql→Head→datum_group;
    if ((_SELECT_FORMAT ← (char *) malloc(strlen(column) + 14)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if (Psql→flags & _USE_GROUP_NAME) (void) sprintf(_SELECT_FORMAT, " WHERE %s='%%s' ", column);
    else (void) sprintf(_SELECT_FORMAT, " WHERE %s=%%lu ", column);
  }
```

This code is used in section 14.

27. For efficiency, we will try to have one command sent to `psql` per group, and hence we will cache in a file the attributes if there is more attributes than *chunk_size*.

On first call, when no cache is already filled for this group, we ask for the number of attributes associated. If there are more than *chunk_size*, we declare a *Cache* and we store the information the image. On remaining calls, we search for an entry valid with the same group.

If the offsets don't match we invalidate the cache.

28. So here is the format of the *Caches* array of *Cache* structures. We will limit the amount of caches open at the same time.

```
⟨ Selection structures 24 ⟩ +≡
#define _NB_CACHES 13 /* minimal value of OPEN_MAX minus 3 standard */
  struct {
    FILE *fp;
    char *pathname;
    id_ktgroup;
    unsigned long image_offset;
    unsigned long multiplicity;
  } Caches[_NB_CACHES];
#define _CACHES_ADMIN→Caches
```

29.

```
⟨ psql/get.c 29 ⟩ ≡
#include "psql/_psql.h"
  int C_psql_get_image(struct C_DFile *Psql, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    unsigned long i;
    int cache;
    char *buf;      /* set by G_fread_line */
    FILE *fp_in;      /* from where we get data */
    unsigned long nb_atts_here;
    char *select_clause ← Λ;
    struct C_Att *Att;
    _CHECK_HANDLER;      /* check mode */
    if (Psql→mode ≡ G_O_WRONLY) return C_EMODE;
    fp_in ← _RESULT;      /* we start with the pipe */
    for (cache ← 0; cache < _NB_CACHES; cache++)
      if (_CACHES[cache].fp ≠ Λ ∧ _CACHES[cache].group ≡ Image→group) break;
    if (cache < _NB_CACHES ∧ _CACHES[cache].image_offset ≠ Image→offset) {
      ⟨ Invalidate the cache entry 39 ⟩
      cache ← _NB_CACHES;
    }
    if (cache ≡ _NB_CACHES) {
      ⟨ Compute the particular select_clause 30 ⟩
      ⟨ Retrieve the multiplicity 34 ⟩
      if (Image→multiplicity > Psql→Head→chunk_size ∧ Image→offset < Image→multiplicity)
        ⟨ Allocate a cache, set cache to the entry and seek to image_offset 38 ⟩
    }
    else Image→multiplicity ← _CACHES[cache].multiplicity;
    if (Image→multiplicity ≤ Image→offset) nb_atts_here ← 0;      /* done */
    else nb_atts_here ← (Image→multiplicity − Image→offset > Psql→Head→chunk_size) ? Psql→Head→chunk_size :
          Image→multiplicity − Image→offset;
    if ((status ← C_alloc_atts(Psql→Data→nb_datums, Image, nb_atts_here))) goto end;
    if (nb_atts_here) {
      if (cache ≡ _NB_CACHES) {      /* can all be done on fly */
        (void) fprintf(_CMD,
            "SELECT␣oid,␣*␣FROM␣%s␣%s␣ORDER␣BY␣oid␣ASC␣LIMIT␣%lu␣OFFSET␣%lu;\n",
            Psql→Head→table, select_clause, nb_atts_here, Image→offset);
      }
      else fp_in ← _CACHES[cache].fp;
      assert(fp_in ≠ Λ);
      for (i ← 0; i < Image→nb_atts_here; i++) {
        Att ← &Image→Atts[i];
        ⟨ Set the fields from line of output; goto end on error 36 ⟩
      }
    }
    free(select_clause);
    if (cache < _NB_CACHES) _CACHES[cache].image_offset += Image→nb_atts_here;
  end: return status;
  }
```

30. Computing the select clause.
The select clause for this group will depend...on the group. This does not only mean take the group name instead of group if this is requested, but to take into account the special groups that might have special select clauses.

```
⟨ Compute the particular select_clause 30 ⟩ ≡
  {
    char *p, *q;      /* used to compute the string length */
    if (Psql→flags & _USE_GROUP_NAME) {
      if (Image→group ≡ P_GROUP_EMPTY ∧ Psql→Head→group_empty_as ≠ Λ)
        ⟨ Compute the select clause for P_GROUP_EMPTY 31 ⟩
      else if (Image→group ≡ P_GROUP_ALL ∧ Psql→Head→group_all_as ≠ Λ)
        ⟨ Compute the select clause for P_GROUP_ALL 32 ⟩
      else ⟨ Get group name and set the select clause 33 ⟩
    }
    else {
      for (p ← _SELECT_FORMAT; *p; p++) ;
      if ((select_clause ← (char *) malloc((size_t)(p − _SELECT_FORMAT) + 13)) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      (void) sprintf(select_clause, _SELECT_FORMAT, Image→group);
    }
  }
```

This code is used in section 29.

31. P_GROUP_EMPTY is used with attributes that have no geometry linked. As for other special cases, if *group_empty_as* is not set (as a SQL WHERE clause for example), the normal case will be used, taking P_GROUP_EMPTY number or its corresponding group name.

```
⟨ Compute the select clause for P_GROUP_EMPTY 31 ⟩ ≡
  {
    for (p ← Psql→Head→group_empty_as; *p; p++) ;
    if ((select_clause ← (char *) malloc((size_t)(p − Psql→Head→group_empty_as) + 9)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    (void) sprintf(select_clause, "␣%s␣", Psql→Head→group_empty_as);
  }
```

This code is used in section 30.

32. the P_GROUP_ALL can be thought as the *dump_all* clause. If the clause is not set in the head, it will be handled via the normal case. This means that if one wants to dump everything, one needs to set the clause as the empty '' one in the head.

```
⟨ Compute the select clause for P_GROUP_ALL 32 ⟩ ≡
  {
    for (p ← Psql→Head→group_all_as; *p; p++) ;
    if ((select_clause ← (char *) malloc((size_t)(p − Psql→Head→group_all_as) + 9)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    (void) sprintf(select_clause, "␣%s␣", Psql→Head→group_all_as);
  }
```

This code is used in section 30.

33. There are several things to do with the group name.

First to retrieve it, and second to escape the single quotes since `psql` will be quite confused if they happened to appear in the middle of the names we are trying to pass to it.

```
⟨ Get group name and set the select clause 33 ⟩ ≡
  {
    struct P_Pair Pair;
    char *_qs;      /* special name used by the escaping club */
    Pair.group ← Image→group;
    if ((status ← P_find_by_group(Psql→Mapping, &Pair))) {
      status ← C_EPFINDGROUPNAME;
      goto end;
    }
    _qs ← &Pair.gname;
    ⟨ Escape single quote 40 ⟩
    for (p ← _SELECT_FORMAT; *p; p++) ;
    for (q ← _qs; *q; q++) ;
    if ((select_clause ← (char *) malloc(((size_t)(p − _SELECT_FORMAT)) + ((size_t)(q − _qs)) + 1)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    (void) sprintf(select_clause, _SELECT_FORMAT, _qs);
    free(_qs);
  }
```

This code is used in section 30.

34. Once the *select_clause* is computed, retrieving the multiplicity is quite simple.

```
⟨ Retrieve the multiplicity 34 ⟩ ≡
  {
    (void) fprintf(_CMD, "SELECT␣count(*)␣FROM␣%s␣%s;\n", Psql→Head→table, select_clause);
    ⟨ Skip blank lines to usefull data in fp_in 11 ⟩
    (void) sscanf(buf, "␣%lu␣", &Image→multiplicity);
    free(buf);
    buf ← Λ;
  }
```

This code is used in section 29.

35. Decision to allocate a cache or not will be done on the total amount of attributes to retrieve at the offset.

If there are less than *Psql→Head→chunk_size* we don't need to, and won't allocate a cache. If there is more, we allocate in every case a cache, even by freeing one if there is no room.

36. Miscellanei routines.

Getting the values from a line of output such as to have the *oid* and all the columns is simple if the values are separated by a well known separator.

```
⟨ Set the fields from line of output; goto end on error 36 ⟩ ≡
  {
    ⟨ Skip blank lines to usefull data in fp_in 11 ⟩
    ⟨ Copy each string between SEPARATOR in the corresponding field 37 ⟩
    free(buf);
    buf ← Λ;
    if (status) goto end;        /* we may have break in loop */
  }
```

This code is used in section 29.

37. The values are printed separated by `SEPARATOR`. So we use two pointers, one after the previous separator, one before the next one, to copy the string between the two. We need to copy since the buffer will be freed.

```
⟨ Copy each string between SEPARATOR in the corresponding field 37 ⟩ ≡
  {
    int j;
    char *bp, *ep;        /* our two pointers, begin and end */
    ep ← buf;        /* begin at the beginning */
    for (j ← 0; j ≤ Psql→Data→nb_datums; j++) {
      bp ← ep;
      for ( ; *ep ≠ Psql→Head→separator ∧ *ep; ep++) ;
      *ep ← '\0';        /* truncate this portion */
      if (j ≡ 0)        /* oid */
        (void) sscanf(bp, "%ld", &Att→id);
      else {
        if (Psql→Data→Datums[j − 1].Logical.type)        /* not C_RAW_L */
          G_strip(bp);        /* remove leading and trailing formatting blanks */
        if (Psql→Head→null_as ≠ Λ ∧ (¬strcmp(bp, Psql→Head→null_as))) Att→fields[j − 1] ← Λ;
            /* redondant */
        else Att→fields[j − 1] ← bp ≡ Λ ? Λ : G_store(bp);
      }
      ep++;        /* skip Λ for loop, harmless at end */
    }
  }
```

This code is used in section 36.

38. Allocation/deallocation of caches.

We can now examine the allocation and deallocation of caches. In every case, we request the allocation of a cache if the multiplicity exceeds *Psql→Head→chunk_size* in order to speed things a bit, specially when retrieving all the attributes not associated with a geometry (`P_GROUP_EMPTY`).

This means that if there is no more cache left, we will invalidate one to take the place. We do this blindly: the first on the list.

But we start by looking for some room.

Please note that the *image_offset* will be updated after: we start at 0 when allocating, and the cache is moved if needed afterwards.

```
⟨ Allocate a cache, set cache to the entry and seek to image_offset 38 ⟩ ≡
  {
    for (cache ← 0; cache < _NB_CACHES ∧ _CACHES[cache].fp ≠ Λ; cache↔) ;
    if (cache ≡ _NB_CACHES) {      /* not found */
      cache ← 0;
      ⟨ Invalidate the cache entry 39 ⟩
    }
    if ((_CACHES[cache].pathname ← G_tempfile( )) ≡ Λ) {
      status ← C_psql_EOPENCACHE;
      goto end;
    }
    _CACHES[cache].group ← Image→group;
    _CACHES[cache].multiplicity ← Image→multiplicity;
    (void) fprintf(_CMD, "\\o␣%s\n", _CACHES[cache].pathname);
    (void) fprintf(_CMD, "SELECT␣oid,*␣FROM␣%s␣%s␣ORDER␣BY␣oid␣ASC;""\\echo␣'DONE\\n'\n\\o\n",
        Psql→Head→table, select_clause);
    for (G_fread_line(&buf, _RESULT); buf ≠ Λ ∧ strcmp(buf, "DONE"); ) {
      free(buf);
      G_fread_line(&buf, fp_in);
    }
    if ((_CACHES[cache].fp ← fopen(_CACHES[cache].pathname, "r")) ≡ Λ) {
      status ← C_psql_EOPENCACHE;
      goto end;
    }
    fp_in ← _CACHES[cache].fp;
    assert(Image→offset < Image→multiplicity);
    for (_CACHES[cache].image_offset ← 0; _CACHES[cache].image_offset < Image→offset;
          _CACHES[cache].image_offset↔) {
      ⟨ Skip blank lines to usefull data in fp_in 11 ⟩
      free(buf);
      buf ← Λ;
    }
  }
```

This code is used in section 29.

39. When invalidating the entries, we need to free the resources.

```
⟨ Invalidate the cache entry 39 ⟩ ≡
  {
    (void) fclose(_CACHES[cache].fp);        /* close the file */
    _CACHES[cache].fp ← Λ;
    (void) unlink(_CACHES[cache].pathname);
    (void) free(_CACHES[cache].pathname);
    _CACHES[cache].pathname ← Λ;
    _CACHES[cache].group ← 0;
    _CACHES[cache].multiplicity ← 0;
    _CACHES[cache].image_offset ← 0;
  }
```

This code is used in sections 22, 29, and 38.

40. SQL idiosyncrasies.

Since SQL requires that not numeric values be enclosed in single quotes, a single quote in the middle of the string will break havoc. So we need to protect them, to escape them by prepending a backslash.

The process is classical. First we realloc one byte more (this is a very conservative approach of memory, but the number of quotes should be limited, and this is simple albeit slow). Them we need to insert a backslash before the quote, this means we need to shift right all the characters after the quote including the quote, and then putting the escape character there in place of the quote.

And the operation needs to be repeated as long as there are quotes... progressing along the string. Hence we need to keep track of the *delta*, that is the number of bytes from the beginning of the string where escaping has to be done next.

This is inlining—this chunk of code is used in various places—, since it is more efficient. We need to define a kind of API: the string to escape is passed via *_qs* (quoted string).

```
⟨ Escape single quote 40 ⟩ ≡
  if (_qs ≠ Λ) {
    char *p, *q;
    size_t delta ← 0;        /* delta from absolute beginning of _qs */
    p ← _qs;
    while ((q ← G_index(p, '\'')) ≠ Λ) {
      char *r;
      delta += q − p;        /* add relative offset */
      for (r ← _qs; *r; r++) ;        /* skip to end for length computation */
      if ((p ← realloc(_qs, (size_t)(r − _qs) + 2)) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      _qs ← p;
      q ← _qs + delta;        /* recompute index q of quote */
      for ( ; *p; p++) ;        /* skip to end (there is a char more after) */
      p++;        /* new end after the "previous" null byte */
      for ( ; p ≠ q; p−−) *p ← *(p − 1);        /* shift one right from end to quote */
      *p ← '\\';
      p ← q + 2;        /* skip forward the new escaped quote */
      delta ← p − _qs;        /* alternatively delta += 2 */
    }
  }
```

This code is used in sections 33 and 49.

41. Table manipulation.

If we are creating a new table, its SQL description must be given. So we simply add to the `_CMD` file the commands, treating datum per datum.

In the same time we will cache the description of the table in *table_desc*.

```
⟨ Convert Data to SQL table 41 ⟩ ≡
  {
    (void) fprintf(_CMD, "CREATE␣TABLE␣%s␣(", Psql→Head→table);
    for (i ← 0; i < Psql→Data→nb_datums; i++) {
      ⟨ Translate datum in SQL 42 ⟩
    }
    (void) fprintf(_CMD, ");\n");
  }
```

This code is used in section 20.

42. Translation of datum in SQL is simple too.

```
⟨ Translate datum in SQL 42 ⟩ ≡
  {
    char *type;
    int precision;
    switch (_DATUMS[i].Logical.type) {
    case C_TEXT_L:
      if ((precision ← _DATUMS[i].Admin.size) < 10) type ← "char";
      else type ← "varchar";
      break;
    case C_LOGICAL_L:
      if (_DATUMS[i].Word.significand > 1) {
        type ← "varbit";
        precision ← _DATUMS[i].Word.significand;
      }
      else {
        type ← "boolean";
        precision ← 0;
      }
      break;
    case C_INTEGER_L: type ← "int";
      precision ← 0;
      break;
    case C_DECIMAL_L: case C_FLOAT_L: type ← "float";
      precision ← 0;
      break;
    case C_DATE_L: type ← "timestamp␣with␣time␣zone";
      precision ← 0;
      break;
    case C_TIME_L: type ← "time";
      precision ← 0;
      break;
    default: G_msg(C_psql_WSETTYPE, C_Warnings, G_WARNING, (unsigned int) _DATUMS[i].Logical.type);
      type ← "varchar";
      precision ← _DATUMS[i].Admin.size;
      break;
    }
    (void) fprintf(_CMD, "%s␣%s", _DATUMS[i].Logical.name, type);
    if (precision) (void) fprintf(_CMD, "(%i)", precision);
    (void) fprintf(_CMD, "%s", (i ≡ Psql→Data→nb_datums − 1) ? "" : ",");
  }
```

This code is used in section 41.

43. We are now able to do the converse operation.

Since the pipe has no `EOF` right now—at least we hope so—and we are requesting an unknown amout of information from PostgreSQL, we need to track the end (with a magic string, flushed because a trailing newline is appended).

```
⟨ Convert SQL table to Data 43 ⟩ ≡
  {
    char *keys[3];      /* will hold name, type, modifiers */
    char *pbuf;         /* temp pointer in buf */
    void *p ← Λ;        /* used when reallocating */
    struct C_Datum Datum ← {{0, Λ}, {0, 1, 0, 1, 0, Λ}, {0, 0, 0}};
    (void) fprintf(_CMD, "\\d %s \\echo 'DONE\\n'\n", Psql→Head→table);
    for (Psql→Data→nb_datums ← 0; ¬G_fread_line(&buf, _RESULT) ∧ buf ≠ Λ ∧ strcmp(buf, "DONE");
          ++Psql→Data→nb_datums) {
      ⟨ Set Datum with the parsed values 45 ⟩
      ⟨ Add a datum 44 ⟩
      free(buf);
    }
  }
```

This code is used in section 20.

44. Adding a datum means reallocating the *C_Data.Datums* array. It must be set to Λ initially.

```
⟨ Add a datum 44 ⟩ ≡
  if ((p ← realloc(Psql→Data→Datums, (Psql→Data→nb_datums + 1) * sizeof(struct C_Datum))) ≡ Λ) {
    status ← errno ≪ G_ERROR_SHIFT;
    free(Psql→Data→Datums);
    Psql→Data→Datums ← Λ;
    goto end;
  }
  Psql→Data→Datums ← p;
      /* nb_datums is still previous value!!! Don't substract 1 */
  Psql→Data→Datums[Psql→Data→nb_datums] ← Datum;
```

This code is used in section 43.

45. We need now to parse each line to retrieve the value. Since the values will be pipe separated, some parts of this code will be used elsewhere.

```
⟨ Set Datum with the parsed values 45 ⟩ ≡
  ⟨ Retrieve the fields in the pipe separated line 46 ⟩
  Datum.Logical.name ← keys[0];
  Datum.Admin.size ← 1;      /* default */
  if (¬strncmp(keys[1], "int", 3)) {
    C_INT32_W(Datum.Word);
    Datum.Logical.type ← C_INTEGER_L;
  }
  else if (¬strncmp(keys[1], "float", 5)) {
    C_IEEE_754_DOUBLE_W(Datum.Word);
    Datum.Logical.type ← C_FLOAT_L;
  }
  else if (¬strncmp(keys[1], "double␣precision", 16)) {
    C_IEEE_754_DOUBLE_W(Datum.Word);
    Datum.Logical.type ← C_FLOAT_L;
  }
  else if (¬strncmp(keys[1], "smallint", 8)) {
    C_INT16_W(Datum.Word);
    Datum.Logical.type ← C_INTEGER_L;
  }
  else if (¬strncmp(keys[1], "bool", 4)) {
    C_SET_W(Datum.Word, 0, 1, 0);
    Datum.Logical.type ← C_LOGICAL_L;
  }
  else if (¬strncmp(keys[1], "timestamp", 9) ∨ ¬strncmp(keys[1], "date", 4)) {
    C_UINT8_W(Datum.Word);
    Datum.Logical.type ← C_DATE_L;
    Datum.Admin.size ← 23;
  }
  else {      /* all texts */
    C_UINT8_W(Datum.Word);
    Datum.Logical.type ← C_TEXT_L;
  }     /* parse the precision */
  {
    int precision;
    if ((pbuf ← G_index(keys[1], '(')) ≠ Λ) {
      (void) sscanf(pbuf, "(%d)%*[^\n]", &precision);
      Datum.Admin.size ← precision;
    }
  }
```

This code is used in section 43.

46. Since the line—due to the parameters passed to `psql`—is pipe separated, parsing is easy.

```
⟨ Retrieve the fields in the pipe separated line 46 ⟩ ≡
  if ((pbuf ← G_index(buf, '|')) ≡ Λ) {
    G_msg(G_ICONTEXT, G_Infos, G_INFO, buf);
    status ← C_psql_EBADFORMAT;
    goto end;
  }      /* buf is freed so name MUST be copied */
  if ((keys[0] ← (char *) malloc(pbuf − buf + 1)) ≡ Λ) {
    status ← errno ≪ G_ERROR_SHIFT;
    goto end;
  }
  (void) strncpy(keys[0], buf, pbuf − buf);
  keys[0][pbuf − buf] ← '\0';
  keys[1] ← ++pbuf;
  if ((pbuf ← G_index(keys[1], '|')) ≡ Λ) {
    status ← errno ≪ G_ERROR_SHIFT;
    goto end;
  }
  pbuf[0] ← '\0';      /* end keys[1] */
  keys[2] ← ++pbuf;
```

This code is used in section 45.

47. For efficiency purpose, things that are constant at open time are cached. This is the case of the description of the table (that may be changed on request in a future version, but this is fixed the most of the time).

When the *Data* is filled with the description, computing the SQL description of the table is straightforward.

```
⟨ Set table_desc 47 ⟩ ≡
  {
    size_t tdl ← 1;      /* to take into account the future trailing T0 */
    char *p;
    unsigned short i;
    for (i ← 0; i < Psql→Data→nb_datums; i++) {
      tdl += strlen(_DATUMS[i].Logical.name) + 3;
      if ((p ← realloc(_ADMIN→table_desc, tdl)) ≡ Λ) {
        free(_ADMIN→table_desc);
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      _ADMIN→table_desc ← p;
      if (¬i)      /* quoting first */
        (void) strcat(_ADMIN→table_desc, "\"");
      else (void) strcat(_ADMIN→table_desc, ",\"");
      (void) strcat(_ADMIN→table_desc, _DATUMS[i].Logical.name);
      (void) strcat(_ADMIN→table_desc, "\"");      /* end quote */
    }
  }
```

This code is used in section 14.

48. Dealing with atts aka records.
Writing an att is always done by inserting a new record and deleting a previous one if it was tagged as modified.

```
⟨ psql/set.c 48 ⟩ ≡
#include "psql/_psql.h"
  int C_psql_set_image(struct C_DFile *Psql, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    unsigned long i;
    struct C_Att *Att;
    _CHECK_HANDLER;
    for (i ← 0; i < Image→nb_atts_here; i++) {
      Att ← &Image→Atts[i];
      if (Psql→flags & C_RESTORING ∨ C_IS_MODIFIED(Att) ∨ ¬Att→id) {      /* restoring, modified or new */
        if (¬(Psql→flags & C_RESTORING) ∧ C_IS_MODIFIED(Att) ∧ Att→id) {      /* delete previous */
          (void) fprintf(_CMD, "DELETE␣FROM␣%s␣WHERE␣oid=%li;\n", Psql→Head→table, Att→id);
        }
        (void) fprintf(_CMD, "INSERT␣INTO␣%s␣(%s%s)␣", Psql→Head→table, Att→id ? "oid," : "",
            _ADMIN→table_desc);
        (void) fprintf(_CMD, "VALUES␣(");
        if (Att→id) (void) fprintf(_CMD, "%lu,", Att→id);
        ⟨ Set every field 49 ⟩
        (void) fprintf(_CMD, ");\n");
        if (C_IS_MODIFIED(Att)) Att→flags &= ∼C_ATT_MODIFIED;
        if (¬Att→id ∨ Psql→flags & C_RESTORING)      /* insertion */
          ++Psql→Data→nb_atts;
      }
    }
  end: return status;
  }
```

49. When inserting new values, we need to enclose strings in quotes, while letting numeric values as is.

```
⟨ Set every field 49 ⟩ ≡
  {
    unsigned long i;
    for (i ← 0; i < Psql→Data→nb_datums; i++) {
      char *_qs;      /* quoted string, use by "inline" escaping procedure */
      _qs ← Att→fields[i] ≡ Λ ? Λ : G_store(Att→fields[i]);
      if (Psql→Head→null_as ≠ Λ ∧ (Att→fields[i] ≡ Λ)) (void) fprintf(_CMD, "%s", Psql→Head→null_as);
      else if (C_IS_STRING(Psql→Data→Datums[i])) (void) fprintf(_CMD, "'%s'", _qs);
      else {
        ⟨ Escape single quote 40 ⟩
        (void) fprintf(_CMD, "%s", _qs);
      }
      (void) fprintf(_CMD, "%s", (i ≡ Psql→Data→nb_datums − 1) ? "" : ",");
      free(_qs);
    }
    (void) fprintf(_CMD, ")");
  }
```

This code is used in section 48.

50. Signal handling.

51. If something happens to the child and the pipe is broken, we need to mark the handler as invalid. Indeed, despite the pipes we will not track the pipes by themselves (we will set the signal indeed to avoid being killed), but we will track the child status to decide what to do.

```
⟨ Set signal handling 51 ⟩ ≡
  {
    struct sigaction act, oact;
    sigaction(SIGPIPE, Λ, &oact);
    if (oact_pipe.sa_handler ≠ _C_psql_sigpipe) {      /* set and save */
      oact_pipe ← oact;
      act.sa_handler ← _C_psql_sigpipe;
      (void) sigemptyset(&act.sa_mask);
      act.sa_flags ← 0;
      if (sigaction(SIGPIPE, &act, Λ) < 0) G_msg(G_WSIGSETFAILED, G_Warnings, G_WARNING);
    }
  }
```

This code is used in section 9.

52. The *_C_psql_sigpipe* is simple: we moan and exit.

This is clearly suboptimal at the moment but since signals are handled by process, we need to find a way to split our gestion from the own of the calling programs (perhaps forking to handle sessions but this will need better IPC).

```
⟨ Private prototypes 52 ⟩ ≡
  void _C_psql_sigpipe(int signo);
```

See also section 54.

This code is used in section 8.

53.

```
⟨ psql/signal.c 53 ⟩ ≡
#include "psql/_psql.h"
  void      /* ARGSUSED */
  _C_psql_sigpipe(int signo)
  {
    G_msg(C_WCLIENTEXIT, C_Warnings, G_FATAL);
  }
```

See also section 55.

54. The *_C_psql_sigchld*, called when some child has exited, waits to identify it and then reset `_PID` in the handler to 0, close the descriptors and the file pointers, but does not free the structure (since a client of the libcorr still have this handler as a valid one—it will be informed that it is no more valid when calling—and the side effect of freeing the structure will be to let the handler identifier ready to be allocated to a new connection.

```
⟨ Private prototypes 52 ⟩ +≡
  void _C_psql_sigchld(int signo);
```

55.

```
⟨ psql/signal.c 53 ⟩ +≡
  void      /* ARGSUSED */
  _C_psql_sigchld(int signo)
  {
    G_msg(C_WCLIENTEXIT, C_Warnings, G_WARNING);
  }
```

56. Index. Here is a list of the identifiers. Overstriked entries indicate the place of definition.