# The Compleat In-Memory Database Designer

Mark Sitkowski
Design Simulation Systems Ltd
www.designsim.com.au

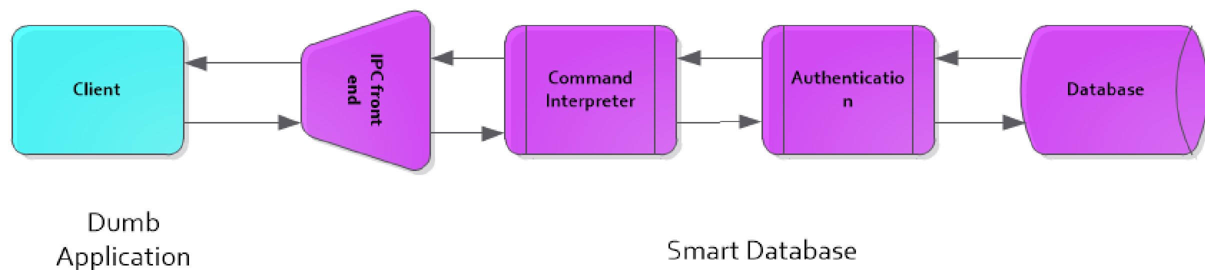
## Introduction

In an earlier paper (“A Child’s Guide to In-Memory Database Design”) we discussed the basics of an In-Memory Database, and showed some skeleton code for the implementation of **sort** and **search**, the two core functions:

In this paper, we build on the principles established there, to produce a complete, working database.

If we had used those code outlines to code a complete, working database, we would have something which worked like a database, and we would probably still be sitting around, wondering what to do with it.

What we would have created, would be something which followed the standard database paradigm.




Any kind of client application can connect to our database and, if it speaks the command language, and can supply the necessary passwords, or whatever, can access the data.

If we assume that the database process is some kind of server, then the sequence of events is as follows:

- accepts a connection,
- reads a command
- figures out what it means
- authenticates the client
- searches for and retrieves the data
- modifies the data, if appropriate (UPDATE, DELETE etc)

Is this really what we want, if we’re not in the business of selling databases?

Let’s say that we just want to design a system which incorporates a database, where an application can access data with the minimum of fuss, and in the shortest time.

Let’s further postulate that we would like the application to be a CGI process which will be the only application access this database.

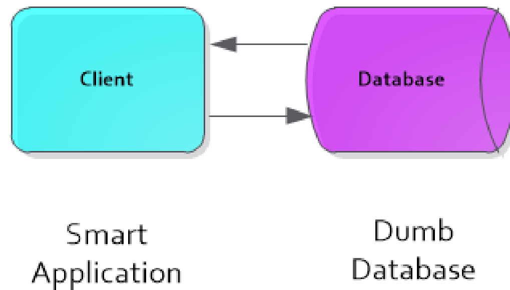
Since our application is a CGI process, we would like to eliminate as many of the above steps as possible, so the sequence of events more closely follows this:

1. CGI process starts
2. It accesses the data
3. It performs any appropriate manipulations of the data
4. It quits.

For reasons of security, we don't want our CGI process to have any kind of dialogue with the database and, especially, we don't want it to need to authenticate itself.

Let's Think Outside The Box.

Why don't we reverse the capabilities of the database and client?




Smart Application

Dumb Database

Our database will store data in memory, which is its sole function. Our client will need to access the data directly, without going through a third party, so why not have the database process create the storage medium in a piece of shared memory, then leave it alone, for the client to access?

Wait.

If the client can directly access the shared memory, then doesn't that mean that anyone can just attach his own process to it, and read it?

No. Our system will, in fact, have the highest level of security that's possible on a Unix system, and here's why.

Certain users, among which 'apache' and 'nobody' are well-known, have no password and no login. The only way a hacker could assume the identity of apache or nobody, would be to login as root, and 'su' to one or the other.
Now, if a hacker has the root password, your whole system is lost anyway so, if apache owns the database, and the CGI runs as apache, we have achieved the highest possible level of security.

In case you're wondering, apache also has to be started by root, to give it the ability to setuid/setgid itself to 'apache', or, 'nobody', if that's the way it was configured.

The disadvantage of shared memory, is that the database must live on the same machine as the client application.
The advantage of shared memory, is that the database must live on the same machine as the client application.

Why?

1. there is no 'Data-In-Flight' vulnerability, as there would be with a network connection
2. most web servers (should definitely) have no user accounts
3. most web servers (should definitely) only run web server software
4. there should definitely be only one person who can login to the server: **root.**

# The Database Design

## Data Structures

Firstly, we need to define data structures to suit, so let's start with the same master table as we used in 'Child's Guide'

```
struct xentry {
   char user_id[25];
   char user_name[25];
   char email[30];
   char app_id[25];
   char app_name[25];
   char url[50];
   char device_id[64];
   char device_name[25];
   char password[64];
   char session_id[64];
   time_t session_start;
 };
struct xentry *entry;
```

(If you missed it, you can download it from here: **www.designsim.com.au/dnlds/ A_Childs_Guide_To_In-Memory_Database_Design.pdf)**

As before, I make no apology for the use of the 'C' programming language for this implementation, since memory efficiency and speed of execution are the prime object of the exercise.
Those with no knowledge of 'C' should close this document and go and design a ruby web page, or something...

Just because we can, let's make another table, by adding another structure, to store information about the system, like its IP address, its name, and couple of other bits. This isn't really necessary, but it illustrates the method for adding additional tables to our database.

```
struct xsettings {
   char licence_key[50];
   char serverid[30];
   char server_url[50];
   char status[10];
   int length;
};
struct xsettings *settings;
```

Now, we're going to define yet another structure, a container for the whole database, the reasons for which will become obvious later:

```
struct xdbase {
   struct xsettings *settings;
   struct xentry *entry;
};
struct xdbase *dbase;
```

## The Memory

We now have the structures for the whole database, so let's get some memory, to keep it all in. Let's say we need to store a million rows, in which case we would set the value of max_users, in the code below, to 1000000.

```
int dbsize = 0;              /* database container */
int setsize = 0;             /* settings table */
unsigned int shmids;         /* settings table */
unsigned int shmide;         /* entry table */

   SHMSIZ = sizeof(struct xentry) * max_users;    /* max size */
   dbsize = sizeof(struct xdbase);
   setsize = sizeof(struct xsettings);

   /* first, the root database structure  */
   if((SHMID = shmget((key_t)IPC_PRIVATE, dbsize, IPC_CREAT | 0600)) <= 0){
      perror("Error obtaining shared memory for xdbase");
      return(-1);
   }

   /* next, the settings structure */
   if((shmids = shmget((key_t)IPC_PRIVATE, setsize, IPC_CREAT | 0600)) <= 0){
      perror("Error obtaining shared memory xsettings");
      return(-1);
   }

   /* Now, the xentry memory segment */
   if((shmide = shmget((key_t)IPC_PRIVATE, SHMSIZ, IPC_CREAT | 0600)) <= 0){
      perror("Error obtaining shared memory");
      return(-1);
   }
```

Okay, we've been given a couple of acres of system memory, and now the reason for defining the dbase structure will become apparent. Without that structure, we won't know where our two tables have been placed in memory.

Here's how we do it:

```
   /* attach memory to hold the dbase structure to the dbase pointer */
   if((dbase = (struct xdbase *)shmat(SHMID, (void *)0, SHM_RND&SHM_PAGEABLE))
== (void *)-1){
      perror("Error attaching to dbase memory seg");
      return(-1);
   }
   /* attach this piece of memory to the settings pointer in dbase */
   if((dbase->settings = (struct xsettings *)shmat(shmids, (void *)0,
SHM_RND&SHM_PAGEABLE)) == (void *)-1){
      perror("Error attaching to settings memory seg");
      return(-1);
   }
   /* attach this piece of memory to the entry pointer in dbase */
   if((dbase->entry = (struct xentry *)shmat(shmide, (void *)0,
SHM_RND&SHM_PAGEABLE)) == (void *)-1){
      perror("Error attaching to entry memory seg");
      return(-1);
   }
```

if you're not familiar with shared memory operations then, rather than waste space explaining the above code, try reading www.designsim.com.au/dnlds/IPC_Shm.pdf which may also still exist as a post on Linkedin.

What? Are you still here? I thought you were designing a web page?

After we run the above code, we can type '**ipcs –mb'**, and get the following information:

| SHMID | KEY | MODE | OWNER | GROUP | SEGSZ |
|---|---|---|---|---|---|
| 16777283 | 0 | --rw------- | apache | staff | 244800 |
| 16777282 | 0 | --rw------- | apache | staff | 400 |
| 16777281 | 0 | --rw------- | apache | staff | 16 |

You will notice several things, immediately,
The first, is that just by typing 'ipcs –mb' you were able to find out the SHMID of each memory segment.
The second, is that the information is of no use to you.
The permissions on all of the above memory segments permit reading and writing by apache, and nobody else. Take another look at the shmget() code above, and notice the '0600' creation mask. That is what sets this security feature,

## Loading The Structures

There's no point in discussing the parser for the input file, since your format may not be the same as anyone else's, so readers are strongly urged to be creative, in designing one. However, my personal preference is for a pipe-separated file, since it permits embedded spaces and commas.
Typically, each line might look like:

|APP|user_id|user_name|application_id|application_name|,,,.etc

Then, it's just a matter of running a pointer, in the usual way, from pipe to pipe along the line, extracting the characters of each field into the appropriate field of the structure member. The choice of which fields are encrypted is also a matter for individual discretion.

The only thing which is important, is that you must count the entries as you read them, and place the total in the 'length' field of the dbase->settings structure. The reason for this is that, when the client connects to the database, it has no other clue as to how many entries it contains. This is especially important, if your design requests the maximum available memory on startup, and only partially fills it with data.

## The Database Process.

We mentioned earlier, that the only function of a database is to store data. From a practical point of view, this means that we will need to create a process which, basically, contains no functions, and does nothing, like this:

```
some_function_to_create_the_shared_memory();
a_parser_to_read_master.dir();
setsignals();

 switch((pid = fork())){
    case -1:
       perror("Fork failed");
    break;
    case 0:
```

```
        if(setsid() == -1){
          printf("Warning: database server can't set pgrp\n");
        }
        pause();
      break;
      default:
        shmdt((char *)0);    /* detach parent from IMDB */
        exit(0);
      break;
    }
```

It can be seen from the above, that we first create our shared memory segments, after which we load the data, and then set our interrupt mask – of which more, later..
Next, we create a daemon. Some purists might argue that this is actually a background process, since we only fork once, and don't close all open file descriptors, but the child process is still adopted by init(), and loses its connection to stdin and stdout, so what it's called is a matter of individual conscience.
As can also be seen. the parent process disconnects itself from the shared memory it created, by calling **shmdt().** This is important, as the kernel won't remove the segment until all links to it are removed.

The child process calls **pause()** then, apparently, sits patiently doing nothing. However, what it is actually doing, is waiting for an interrupt. If you've forgotten all you ever knew about interrupt handlers, try this : www.designsim.com.au/dnlds/Signals.pdf

One function of our interrupt handler should be to ignore signals like control-C, to prevent someone from accidentally crashing the process.

A more useful consideration, is that it may be necessary to re-read the input file, in the event that, for example, we need to manually add a new block of users, or delete a few.
Lastly, if our system permits on-line manipulation of the database contents, we will need a function to dump the database to a file (preferably in the same format as the file we read), to save the changes.

Both your re-read and the dump functions can be accessed by setting actions in the signal handler, associated with chosen signal numbers. SIGUSER1 and SIGUSER2 come to mind.

## Access By Client

Now that we have our dumbed-down database running, our CGI client needs to connect to its shared memory segments.
First, we will need our CGI code to contain the definitions of the database structures and, second, it will need knowledge of the three SHMID's that were allocated by the kernel to the database.
Remembering that these SHMID's are of no use to an attacker, we don't need to do anything elaborate, like encryption, so putting them into a file is perfectly adequate – apache does this with its own PID, with no ill-effects.
Our CGI starts up, reads the file, then does exactly the same as the database did:

```
  /* attach ourselves to dbase structure   */
  if((dbase = (struct xdbase *)shmat(SHMID, (void *)0, SHM_RND&SHM_PAGEABLE))
== (void *)-1){
      perror("Error attaching to dbase memory seg");
      return(-1);
  }
  /*  attach settings to dbase   */
```

```
    if((dbase->settings = (struct xsettings *)shmat(shmidx, (void *)0,
SHM_RND&SHM_PAGEABLE)) == (void *)-1){
        perror("Error attaching to settings memory seg");
        return(-1);
    }
    /*  attach entry to dbase   */
    if((dbase->entry = (struct xentry *)shmat(shmide, (void *)0,
SHM_RND&SHM_PAGEABLE)) == (void *)-1){
        perror("Error attaching to entry memory seg");
        return(-1);
    }
```

Now our client is securely connected to a secure database, and can run some of its in-built commands,

## An Advanced Search Function

In the Child's Guide, we gave a simple example of a comparator function, to be used by qsort(). The code shown at the end of this paper sorts on user_id, app_id and device_id.

We also outlined (very) sketchily, how a binary search function would work. However, the function described has the drawback that it only searches for the three hard-coded structure elements passed to it. This is fine, if we only ever wanted to find entries defined by a unique combination of user_id, app_id and device_id. However, if we wanted to search for all users of a particular application, then, we would need a separate function. If this is not be a problem, feel free to ignore what follows.

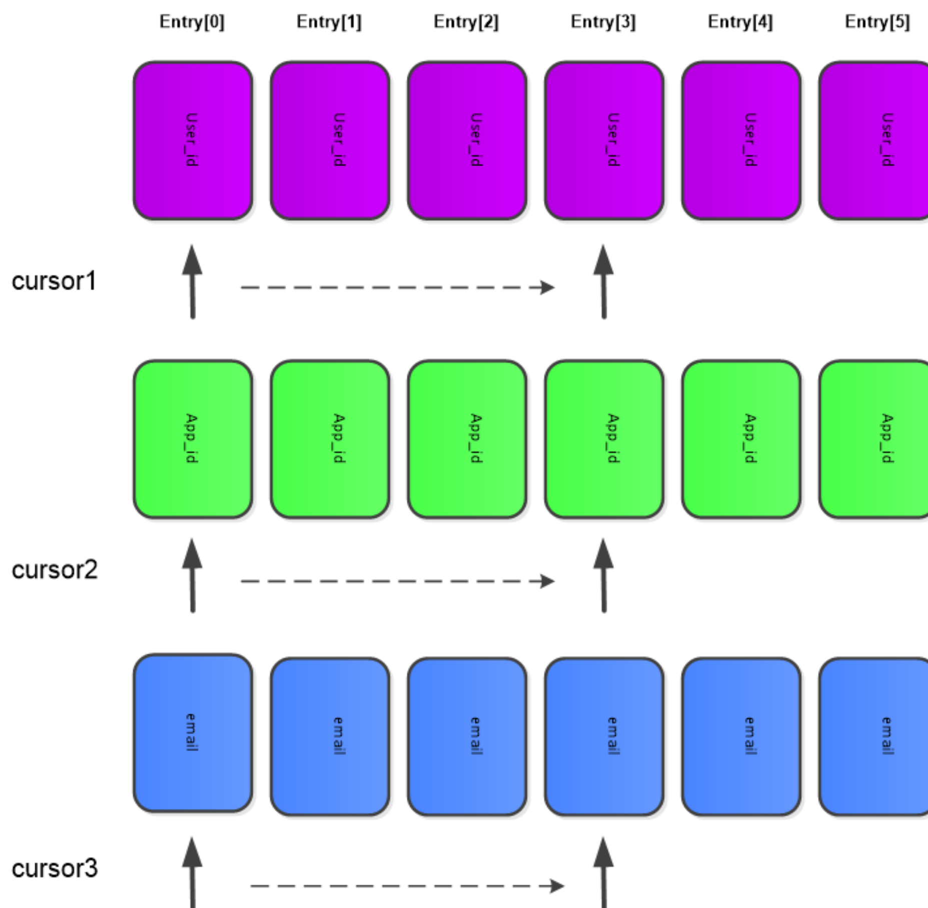
In order to make a generic search function, we will need to pass in not only the variables themselves, but where to find them. For example, if we wanted to search for an email address, we would need to call the function like

search("joebogg@email.com", dbase->entry[0].email);

The search() function now sees the second argument as a pointer to a character array passed to it. Incrementing the pointer will merely make it see the next character in the array, which means that we can no longer search the database by array index.

Instead, we define an increment as the sizeof one xentry structure, which is the sum of all the members, and will be 405 bytes for the structure definition shown earlier.

Next, we define one pointer (which we'll call a 'cursor') to each of the structure members for which we are searching, and initialise each one, by pointing it at the appropriate array in dbase->entry[0]. Then, as the loop increments, we increment each pointer, by adding 405 to its address.
Hopefully, the diagram below will help to visualise how this works.




The function shown below has had all of its error-checking code removed to make it more compact. It must be passed three search strings, tgt1, tgt2 and tgt3 like

**“fred123”, “music_portal”, “dell_laptop_id”**

and three pointers to the structure members sch1, sch2 and sch3, containing the appropriate data, like

**dbase->entry[0].user_id, dbase->entry[0].app_id, dbase->entry[0].device_id**

It returns the array index of the structure it found on success, and -1 on failure.

```
search(tgt1, tgt2, tgt3, sch1, sch2, sch3)              /* search */
char *tgt1;                   /* primary search string */
char *tgt2;                   /* secondary search string */
char *tgt3;                   /* tertiary search string */
unsigned char *sch1;
unsigned char *sch2;
unsigned char *sch3;
{
int flag = -1;                /* found/not found */
int ret;                      /* strcmp’s return */
unsigned char *cursor1;
unsigned char *cursor2;
unsigned char *cursor3;
```

```
unsigned int start;            /* current search index */
unsigned int j;                /* secondary search index */
unsigned int step;             /* increment */
unsigned int pleft = 0;
unsigned int pright = 0;
unsigned int incr = sizeof(struct xentry);
unsigned int dbs = 0;                         /* first element */
unsigned int dbe = incr * lentry;             /* last element */
unsigned int dbsize = dbe - dbs;              /* address range */
unsigned int temp = 2;

   step = dbe / temp;              /* start in the middle */
   cursor1 = sch1;                 /* initialise */
   cursor2 = sch2;                 /* initialise */
   cursor3 = sch3;                 /* initialise */

   cursor1 += step;                /* start in the middle */
   cursor2 += step;                /* start in the middle */
   cursor3 += step;                /* start in the middle */

   start = step;                   /* address from which we search */
   pleft = start;

   while(start >= dbs && start <= dbe){
      if(step == incr) step = (incr * 2);

      ret = strcmp(tgt1, cursor1);

      if(ret > 0){                 /* match is to the right */
         temp *= 2;
         step = (lentry / temp) * incr;
         start += step;       /* to the right of middle */
         cursor1 += step;     /* to the right of middle */
         cursor2 += step;     /* to the right of middle */
         cursor3 += step;     /* to the right of middle */

         if(start == pright){
            printf(">%s< not found\n", tgt1);
            return(flag);
         }

         pright = start;
      }
      else if(ret < 0){            /* match is to the left */
         temp *= 2;
         step = (lentry / temp) * incr;
         start -= step;          /* to the left of middle */
         cursor1 -= step;     /* to the left of middle */
         cursor2 -= step;     /* to the left of middle */
         cursor3 -= step;     /* to the left of middle */

         if(start == pleft){
            printf(">%s< not found\n", tgt1);
            return(flag);
         }
```

```
        pleft = start;
      } else {                            /* got a match on primary search target */
        if(strcmp(cursor2, tgt2) == 0 &&
          strcmp(cursor3, tgt3) == 0){      /* matched all three */
          flag = start / incr;
          return(flag);
        }
        j = incr;                         /* secondary search to the left of here */
        if(cursor1 > sch1){
          while(strcmp((cursor1-j), tgt1) == 0){
            if(strcmp((cursor2-j), tgt2) == 0 &&
              strcmp((cursor3-j), tgt3) == 0){
                flag = (start - j) / incr;
                return(flag);
            }
            if(cursor1-j <= sch1){
              break;
            }
            j += incr;
          }
        }
        j = incr;                        /* secondary search to the right of here */
        if((start + incr) < dbe){
          while(strcmp((cursor1+j), tgt1) == 0){
            if(strcmp((cursor2+j), tgt2) == 0 &&
              strcmp((cursor3+j), tgt3) == 0){
               flag = (start + j) / incr;
               return(flag);
            }
            j += incr;
            if(j >= dbe) break;
          }
        }
        break;
      }
    }
    return(flag);
```

## ORDER BY user_id, app_id, device_id

This function was described, albeit sketchily, in the Child's Guide. The extended version is shown below.

```
int
cmpentry(p1, p2)              /* cmpentry */
const void *p1, *p2;
{
struct xentry *q1, *q2;
    q1 = (struct xentry *)p1;
    q2 = (struct xentry *)p2;
    if(strcmp(q1->user_id, q2->user_id) == 0){    /* matched user_id */
      if(strcmp(q1->app_id, q2->app_id) < 0){
        return(-1);
      }
```

```
        else if(strcmp(q1->app_id, q2->app_id) > 0){
            return(1);
        } else {                           /* also matched app_id */
            if(strcmp(q1->device_id, q2->device_id) < 0){
                return(-1);
            }
            else if(strcmp(q1->device_id, q2->device_id) > 0){
                return(1);
            } else {                       /* also matched device_id */
                return(0);
            }
        }
    } else {
        if(strcmp(q1->user_id, q2->user_id) < 0) return(-1);
        else if(strcmp(q1->user_id, q2->user_id) > 0) return(1);
        return(0);
    }
}                       /* cmpentry */
```

The ADD, DELETE, UPDATE functions were also covered in the Child's Guide, but are repeated here for convenience.

Given a search function, these functions are trivial to code, (unless you're the guy who should be designing a web page...) and are left as an exercise for the reader. The implementation details which follow may act as a useful hint.

## Deleting an Entry

The list is ordered in ASCII order, which is the key to how we delete entries. The method is as follows:

- Search the list for the entry to be deleted
- Overwrite the user_id field with "zzzzzz"
- Sort the list
- Trash the last entry, which must be the one with user_id = "zzzzzz"

## Adding an Entry

This is a trivial operation, provided the memory allocated at startup was greater than that required to hold the data. For a dynamically changing list, the recommended practice, is to shmget() the maximum memory i.e enough for one million entries. Then:

- Increment the 'length' member of the dbase->settings structure to the new list length.
- Write the data to the last entry
- Sort the list

## Updating an Entry

This is too obvious:

- Search the list for item to be updated.
- Update data.

## Concurrency

You don't have to be an expert in Petri nets to see that such a database has unlimited concurrency, if it is treated as a read-only resource (actually, the limit is the number of open file descriptors permitted by the system).

However, when an INSERT or DELETE operation is performed, the data needs to be re-sorted, which will alter the length of the list. This will totally trash any results obtained by a concurrent query performing a search.

Oracle, DB2 and others employ row locking, to prevent queries from reading past rows that are actively being modified. This results in a delay until the query completes. Such an approach is possible if the active query is an UPDATE, but will not help if a table sort is running.
What we need is the equivalent of a mutex, to delay access to the database by incoming queries, until the current one completes. A qsort of one million rows takes between one and two seconds, so the overhead is not so alarming, especially as 99% of accesses will be of the SELECT variety.
The way the mutex is implemented, is to use the permissions on the file containing the SHMID's as a semaphore. It's obviously possible to use real semaphores, and the user is encouraged to explore ways of identifying and resetting them after a system reboot.

When our database has been allocated its requested memory, it deletes the existing SHMID file, and creates a new one, with permissions 600. This happens with every startup, so we always have a known starting point.
When the CGI starts up, it first checks the permissions on the file. If the permissions are 600, it proceeds with the access but, if it sees 400, it waits until they change. If the proposed operation is an INSERT or a DELETE, the CGI sets the permissions to 400, resetting them to 600 after it has performed its qsort().

An alternative, is for the administrator to be solely responsible for all INSERT and DELETE operations, which he performs with the system offline.

## Conclusion

You include the search(), upd_entry(), del_entry() and cmpentry() functions in the CGI code, and you have all the smarts needed to access the database.
Writing the actual database server is accomplished with a few cut'n'paste operations, coupled with writing your file parser and dump function.

## Disclaimer

All code shown was cut and pasted from a working application, with a few minor changes to make it more generic. It's extremely possible, that some bits fell off in the process, so feel free to complain to me, and I may correct any errors.