A Child's Guide to In-Memory Database Design

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

Introduction

A few years ago, an undergraduate in India sent me an email, saying she needed to design a directory server as part of her Master's thesis, but didn't know how to write code. What follows is the essence of what I sent her.

These design techniques can be applied to directory servers, LDAP servers, and to the synthesis of analogues of simple relational databases.

We will not implement any kind of SQL parser or interpreter, since one of the expected advantages is speed of response, which would not be helped by a cumbersome SQL processor, sitting like a bottleneck at the front end.

Also, we will limit the functionality to the implementation of the main tasks expected of a database, namely:

- SELECT ... FROM ... WHERE...
- INSERT INTO ....
- UPDATE ... SET ...
- DELETE FROM ... WHERE ...

In the case of SELECT and DELETE, the predicate will be limited to NAME = VALUE.

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 find something else to do.

Specification for Sample Database

We will assume that our database contains data related to the access, by a number of users, to a series of applications, or 'apps' as it is now trendy to call them.
This suggests two tables: one containing user details, and the other containing application details. Further, we will assume that we wish to keep details of any device that each user has, for accessing the app. That makes table number three.

We can see that the relationship between users and apps is one-to-many, since a given user can be registered to more than one bank account, music download portal, shopping site, or whatever.
We can also see that the relationship between the user and the device is also one-to-many, since a given user can have many devices.

To add a few numbers to the specification, let's say that we wish to cater for a million users, each wishing to make use of any number from one to ten applications. Each user can do so using either Firefox, Chrome, Opera, Edge. Safari or Pale Moon, running on either a laptop,

a smartphone, a tablet or a desktop. This makes a total of twenty-four devices for which we should cater.

Data Structures

Each table can be represented by a data structure, with the elements of the structure representing the columns, while the rows of the table are represented by instances of the structure.

There are two obvious ways to implement the tables. We can either make a linked list, or we can create an array of structures. Both implementations have advantages and disadvantages, which we'll discuss later.

Let's start by naively specifying the Users table, the Application table and the Devices table, and declaring a pointer to each, for the purpose of, later, creating a list of some sort.

```
struct xusers {
    char user_id[25];
    char user_name[25];
    char email[30];
};
struct xusers *users;

struct xapp {
    char app_id[25];
    char app_name[25];
    char url[50];
};
struct xapp *app;

struct xdevice {
    char device_id[64];
    char device_name[25];
};
struct xdevice *device;
```

This arrangement looks okay but we haven't yet considered the idea of session control, or passwords, so let's give it some thought.

So, what's a session, anyway? In essence, it's the connection, by the user, to an application, using one device and one password.

If we put the password into the Users table, we'll need an extra instance of the Users structure for every application to which the user is subscribed. If we put the password into the Apps table, we'll need an extra instance of the Apps structure for every user subscribed to it.
What if we put it in the Device table? Each device connects to one or more applications, so we'll need an extra instance of this structure per subscribed application. This sounds like a session, so we can add session parameters to the device, to indicate an active connection to an application.

That sounds good, but how do we relate the device, to the application, to the user?
Whether we store these tables as a pair of linked lists, or a pair of structure arrays, we can define the relationship between a user, an application and a device, by setting a pointer from

a given app structure to the appropriate users structure or device structure, so our new structures would look like:

```
struct xusers {
    char user_id[25];
    char user_name[25];
    char email[30];
};

struct xapp {
    char app_id[25];
    char app_name[25];
    char url[50];
    struct xusers *user;
    struct xdevice *device.
};

struct xdevice {
    char device_id[64];
    char device_name[25];
    char password[64];
    char session_id[64];
    time_t session_start;
 };
```

Now, by examining the app structure, we can tell who is registered to it, and by which device. By following the device pointer, we can get the time the last session started, what the session_id was, and which device was used. The pointer to the users structure gives us the user details.

Having got this far, let's think about the numbers.

Each Users structure is 80 bytes. Each device is worth (64 + 25 + 64 + 64 + 8) = 225 bytes, and each App is now
(25 + 25 + 50 + 8 + 8) = 116 bytes.

Since we're catering for 1 million users, the minimum configuration, where each user is registered to one application, accessed through one device, will need (80 + 225 + 116) = 421 MB of RAM. This means that we have the equivalent of 1 million rows in each table.

The reality, of course, is that each user will probably have registered each one of his devices to each app to which he was subscribed. This means that the devices table will have 24 million rows, made up as (225 * 24) = 5400 bytes per registered application.

Now, our total RAM requirements have increased, somewhat, to (80 + 5400 + 116) = 5596 MB for our one million users, each registered to one application, and all 24 devices.

By extrapolating to a maximum of five applications per user, we get our final figure of (5 * 5596) = 27980 MB, which is about 28 GB, which is well within the 64 GB, typically present in the average top-end server.

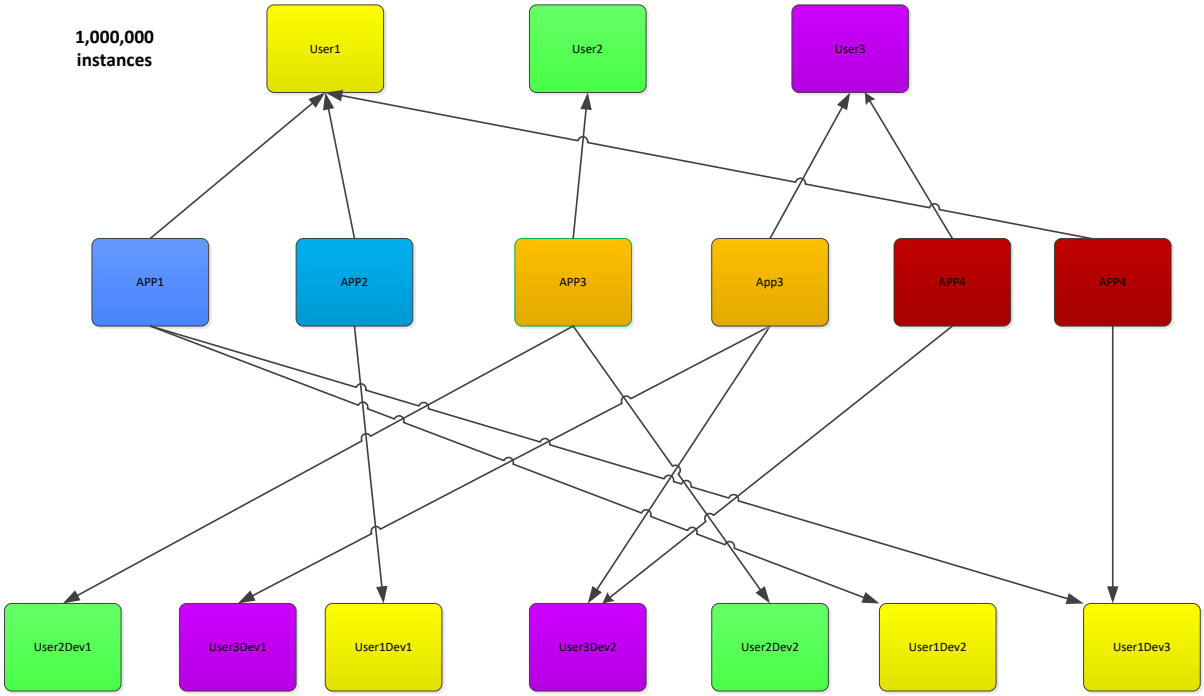
The above exercise underlines the fact that the accurate specification of each data structure is critical, since every byte translates to a megabyte in the final design.

To make things clear, here is a representation of the database after 3 users have been entered, and subscribed to 4 applications. User1 is subscribed to App1, App2 and App4, using his Dev1 device for App2, his Dev2 device and his Dev3 device for App1.

Users 2 and 3 are subscribed to App3, User2 using his Dev1 and Dev2 devices, while User3 uses his Dev1 and Dev2 devices to connect to it.

User3 is also subscribed to App4, and connects with his Dev2 device.

Any relationships not explained above, are left as an exercise for the reader.



List Architecture

The question of list architecture should now be considered, but needs to be examined in the light of what we finally intend to do.

We stated earlier, that we wish to implement SELECT, INSERT, UPDATE and DELETE, all of which imply searching the list, to find the appropriate element on which to operate.

All search algorithms need a sorted list to be provided as input, so let's consider the effects of sorting the list shown above.

Look at the Application array. All the pointers have their tails in the data structure. If we sort this array, the addresses of the pointer targets will follow the elements, and we'll have a sorted Application list, without losing the locations of elements in the Users list, or the Devices list.
On the downside, we'll have to do a linear search of the Users array (one million entries) and the Devices array (24 million entries).

This is no good.

We've been looking at the design from a database designer's point of view, instead of that of a programmer, and need to think more about the functionality and less about tables.

What is it we're actually after? When the database is running, it will be creating and destroying accesses by users to applications. In other words, our design should be session-orientated.

When we have our maximum number of users, registered to the maximum number of applications, with the maximum number of device, what are the worst-case sizes of the tables we created?
For 1 million users, each registered to 5 applications, through 24 devices, we get:

Users	1,000,000
Applications	5,000,000
Devices	24,000,000

Why don't we amalgamate our tables, and just have one structure per entry:

```
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;
};
```

Our new structure requires (25 + 25 + 30 + 25 + 25 + 50 + 64 + 25 + 64 + 64 + 8) = 405 bytes. In order to accommodate the maximum number of users, applications and devices, the array of new structures will contain 24 million elements Memory requirements will be (405 * 24000000) = 9,720,000,000
We've saved nearly 20 GB of RAM, by not having separate tables. Also, we have no pointers to get lost, and we can think seriously about the list structure.
As any child will tell you, sorting, adding to or deleting from a double-linked list requires the manipulation of two pointers per element, per operation.

On the other hand, if we just use an array of structures, everything becomes much simpler.

We initialise our list by creating a dummy element, in which we can keep information about the list:

```
structsize = sizeof(struct xentry);
struct xentry *dbentry;
struct xentry *temp;

   if((dbentry = (struct xentry *)malloc(structsize)) == NULL){
      printf("Memory allocation error\n");
      return(-1);
   }
```

Now, we can add elements very simply, like this:

```
if((temp = (struct xentry *)realloc(dbentry, ((k+1) * structsize))) == NULL){
    printf("Can't reallocate memory\n");
}
dbentry = temp;
```

where ‘k’ is the current length of the list.

Depending on how much adding and deleting is anticipated, it may be advantageous to malloc the full 1 million elements at startup, and merely update the details in each element. Indeed, most directory servers will load the contents of a master.dir file at startup, making this an attractive proposition. However, for the purpose of this exercise, we’ll assume that the list grows organically, as new users appear and are registered.

Sorting the List

Since we know that we’re going to eventually need a sorted list, let’s give that some thought.

We’ll use qsort() to do the sorting, since it’s at least an order of magnitude faster than an SQL ORDER BY query, and we only need to write a simple comparison function to make it work. The comparison function just needs to return a 0 if the compared elements are equal, -1 if the ASCII value of one is less than the other, or +1 if it’s greater. For this reason, the thing that actually does the comparison, is almost always strcmp().

Too easy:

```
Int cmpui(p1, p2)                    /* cmpui */
void *p1, *p2;
{
struct xentry *q1, *q2;

    q1 = (struct xentry *)p1;
    q2 = (struct xentry *)p2;
    if(strcmp(q1->user_id, q2->user_id) < 0) return(-1);
    else if(strcmp(q1->user_id, q2->user_id) > 0) return(1);
    else return(0);
}                                    /* cmpui */
```

This function is called by qsort, which we call like this:

```
qsort((void  *)dbentry,  lentry, sizeof(struct xentry), cmpui);
```

where lentry is the length of our list, and dbentry is the pointer to its head, that we derived above.

Now, the above function sorts the whole list on user_id, which is only half the story. If our database had a primary key, it would be a compound key of user_id and app_id, so we’ll need to modify the function.
We need to sort on user_id as above but, having found a match for this, we need to get qsort to turn its attention to the app_id, like this:

```
Int cmpentry(p1, p2)                    /* cmpentry */
void *p1, *p2;
{
struct xentry *q1, *q2;

   q1 = (struct xentry *)p1;
   q2 = (struct xentry *)p2;
   if(strcmp(q1->user_id, q2->user_id) == 0){
      if(strcmp(q1->app_id, q2->app_id) < 0) return(-1);
      else if(strcmp(q1->app_id, q2->app_id) > 0) return(1);
      else 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 observant reader will have noticed that the above function should also sort on device_id. Think of it as a homework assignment.

Now we have a sorted list, so we can select a search function.

Searching the List

We tested the search algorithms shown in the table below, since we considered them to be the most promising candidates. Readers not sharing this opinion are encouraged to investigate their favourite algorithms.

The test was very simple. We recorded the time each algorithm took to perform a Cartesian join i.e searching one array for the occurrence of each row, in turn, of another array. When run on a one million element array, on a mediocre computer, the algorithms gave the following results:

Method	Setup (seconds)	Run (seconds)
Linear Search	0	2200
Knuth's Binary Tree Search	2	3
Hash Table	200	200
Binary search	0	2
Knuth's linear search	12500	12500

- The Linear Search was a naive full array scan, which terminated upon a successful match of each element
- Knuth's B-Tree algorithm used the tsearch() / tfind() functions supplied with most Unix systems
- A hash table was created using hcreate(), and searched with hsearch()
- The Binary Search used a simple 'Divide-and-Conquer' algorithm
- Knuth's linear search was conducted using the lsearch() and lfind() functions supplied with most Unix systems. There may be better implementations of these functions.

The binary search algorithm, which we chose, works like this:

We split the array in half, and check the entry on the left, and that on the right. If the left entry is < than the target, our match will be in the right section. We then halve the array on the right, and repeat the procedure.
For obvious reasons, if the initial check shows that the left entry is > the target, we do this trick with the left half of the array.
When we get a match, we break out of the loop and return.

The search pattern is exponential, which is why it's so fast.

Such a search function, which implements the equivalent of
SELECT session_id FROM table
WHERE user_id = :userid AND app_id = :applicationid AND device_id = :deviceid;
may be coded as follows:

```
search(userid, applicationid, deviceid, sessionid)                    /* search */
char *userid;
char *applicationid;
char *sessionid;
{
int i;              /* current search index */
int step;           /* search increment */
int flag = 0;       /* found */
int ret;            /* strcmp return */

i = step = lentry / 2;                    /* lentry is length of array */
   while(i >= 0 && i <= lentry){
      if(step == 1) step = 2;
      ret = strcmp(userid, dbentry[i].user_id);
      if(ret > 0){          /* match is to the right */
         step /= 2;
         i += step;
        if(i == pright){
            printf(">%s< not found\n", userid);
            return(flag);
         }
         pright = i;
      }
      else if(ret < 0){       /* match is to the left */
         step /= 2;
         i -= step;
         if(i == pleft){
            printf(">%s< not found\n", userid);
            return(flag);
         }
         pleft = i;
      } else {                /* got a match */
         flag = 1;
         printf("Matched %s <-> %s\n", dbentry[i].user_id, userid);
         /* insert code shown below to find app_id and device_id */
         break;
      }
     return(flag);
}                             /* search */
```

To keep the above code easy to follow, the code to insert for finding the app_id and device_id is shown below.

Basically, when we've found our user_id, we start a search to the left for a matching app_id. If we match that, we check the device_id.

Since we don't know which of many entries matching our user_id we have found, we then do another search to the right of where we are, and again check app_id's and device_id's.

When we find all three, we extract the session_id and copy it to the sessionid parameter passed to the function.

```
j = i;
if(j >= 1){
   while(strcmp(userid, dbentry[--j].user_id) == 0){
      if(strcmp(dbentry[j].app_id, applicationid) == 0){
         if(strcmp(dbentry[j].device_id, device_id) == 0){
            flag = 2;
            strcpy(session_id, dbentry[j].session_id);
            break;
         }
      }
      if(j <= 0) break;
   }
}
j = i;
if(j < lentry){
   while(strcmp(userid, refentry[++j].user_id) == 0){
      if(strcmp(dbentry[j].app_id, applicationid) == 0){
         if(strcmp(dbentry[j].device_id, device_id) == 0){
            flag = 2;
            strcpy(session_id, dbentry[j].session_id);
            break;
         }
      }
      if(j >= lentry) break;
   }
}
```

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 was partially covered earlier, in the section on List Architecture.

- Call realloc() with the new list length. An extra structure is appended to the list
- Write the data to the last entry
- Sort the list

Updating an Entry

This is rather obvious:

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

Conclusion

This is not the only technique by which an In-Memory Database can be designed. However, it is simple and reliable,

Such a database may be incorporated in a daemon, which may accept connections via TCP/IP or across a shared-memory bridge from another process. The code is compact enough to be incorporated into an actual application.

The master.dir input file would probably be encrypted, and decryption software added to the database, together with code to dump the current contents and recreate such a file.

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.